Titulació:
Grau en Enginyeria en Tecnologies Industrials

Alumne (nom i cognoms):
Raúl García Gómez

Enunciat TFG / TFM:
Study for the realization of an acquirement system of high frequency data
(Estudi per la realització d'un sistema d'adquisició de dades d'alta freqüència)

Director/a del TFG / TFM:
Jordi Sellarès Gonzalez

Convocatòria de lliurament del TFG / TFM:
10/6/19

Index

1. Introduction

1.1 Abstract

This project aimed to create a program that would save initial settings of an oscilloscope, give the ability to control it remotely and extract and save data in the context of an actual Optics laboratory experiment with the minimum expense possible. To do so a communication port, code programming and a graphical interface would be needed. These problems were solved with USBTMC, Python and WxGlade. The experiment was carried out with the developed program being able to store initial settings, remotely control the oscilloscope and with data extracted and saved with a Raspberry Pi. The program met the minimum success requirements set up in the objectives, even if it has room for improvement.

1.2 Honour declaration

I declare that,
the work in this Degree Thesis is completely my own work,

no part of this Degree Thesis is taken from other people's work without giving them credit,

all references have been clearly cited,

I'm authorised to make use of the research group related information I'm providing in this document.

I understand that an infringement of this declaration leaves me subject to the foreseen disciplinary actions by *The Universitat Politècnica de Catalunya - BarcelonaTECH*.

Raúl García Gómez	__Raúl__	_10/6/19_
Student Name	Signature	Date

Title of the Thesis :
__Study for the realization of an acquirement system of high frequency data
(Estudi per la realització d'un sistema d'adquisició de dades d'alta freqüència)__

1.3 Project objectives

The objectives of this project are:

- Developing software to store data obtained from an oscilloscope in a computer.

- Developing software with the ability to communicate and remotely operate a oscilloscope.

- Developing software that can change an oscilloscope settings and return to the state before altering them.

- Compiling all of the necessary software into a single program.

- Creating a graphical interface to support and make easier the use of the compiled program.

- Keeping the replication costs of the project to the minimum.

1.4 Project scope

The project's scope includes:

- Schematics and realization of an auxiliary circuit board to be used before testing the program in the actual experiment.

- Code for digital input/output of data gathered by the oscilloscope into the computer.

- Code for remote control of Tektronix digital oscilloscopes to simplify its operation.

- Graphical user interface for integration of the operation of the different elements of the setup.

- Tests in actual experiments.

The project's scope does not include:

- Elaborate data treatment. Because the particulars of the data sought cannot be known (is the user only interested in the amplitude? Or maybe the frequency or only the transitory part of the signal?) doing another program for data treatment would be overreaching and not feasible.

- A normal program format. To explain, due to the nature of the program and the time constraints, creating a "normal program" which in this case I am describing a program that the user only needs to worry about clicking an executable (.exe) is not possible. The user will need to obtain the programming language and the extra libraries needed to use the program before using it, saying in another way, the program cannot be a "download and click" all-in-one program.

1.5 Project requirements

For the project to be successful we will need at least these points to be achieved:

- Guaranteed safety for the components involved in the setup and for the users of the setup. After all this is a project that aims to be used in laboratories to help with practical experiments, factors that could cause harm or danger must be avoided.

- Convenience and ease of use. The program must be another tool to help in experimentation, not another variable or unknown that must be studied.

- Compatibility with an as large as possible list of oscilloscope models. That way no matter what oscilloscope the laboratory uses, the program will remain a useful tool for it.

- Should work on a Raspberry Pi single board computer. Raspberry Pi is specified because of its small size and cost which make it a very useful tool with a high cost/efficiency index.

- The program has to be based on the programming language Python. This is so because Python offers a lot of utility that will be helpful in the creation of the program.

1.6 Project utility

University laboratories usually host a moderate number of researchers and many different experiments are being conducted throughout time with limited material. This material is used by several researchers in different ways and, therefore, to speed up the initial phase of setting up the instruments, this project aims to create an application that allows a user to pre-create oscilloscope settings, control an oscilloscope and retrieve the data and the current settings from the oscilloscope. The application should also be able to remotely control other elements using digital input/output.

To test the application, it will be used to control an experiment involving the measurement of a current obtained by shooting a pulsed laser on a sample. If successful, it will speed up experimentation and make it easier for different researchers to share the lab equipment.

The critical elements of this project will be the communication between computer and the experiment lab apparatus, the design of a convenient graphical user interface, compatibility with commonly available equipment and being able to guarantee the safety of the used resources and of the users during the execution of the application.

2. Development

2.1 State of the art

- LabVIEW: LabVIEW is a paid graphic programming developed by National Instruments. National Instruments is an 80s founded American enterprise with headquarters in Texas and is one of if not the biggest company specialised in electronic data acquisition. The develop both hardware and software for this field, form oscilloscope and sensors to programs like Multisim and LabVIEW. In this case we shall focus on LabVIEW.

  Apart from being paid, the many features and the graphical component it includes makes the user go through a learning phase and to remotely configure the hardware's settings it is necessary to install other plugins. Moreover it is designed with only National Instruments hardware in mind so it is not an interesting solution in this case.

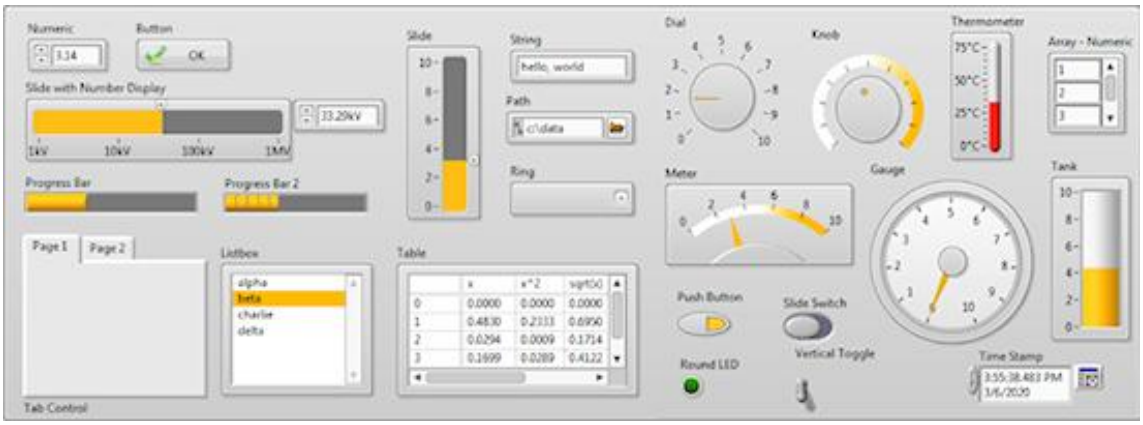

Figure 1: LabVIEW screenshot.

- Tektronix Openchoice Desktop Application: Tektronix Openchoice Desktop Application is also a company related software, this time to Tektronix. Tektronix is also an American company, founded in the 40s, that is part of the Dahaner Corporation and has its headquarters in Oregon. They mainly develop hardware

like oscilloscopes, logic analysers and video and mobile test protocol equipment, but this time the interest fall upon their Openchoice software.

The program is free but suffers from the same limitations as LabVIEW, namely learning curve and limitation to Tektronix's hardware as the most important ones.

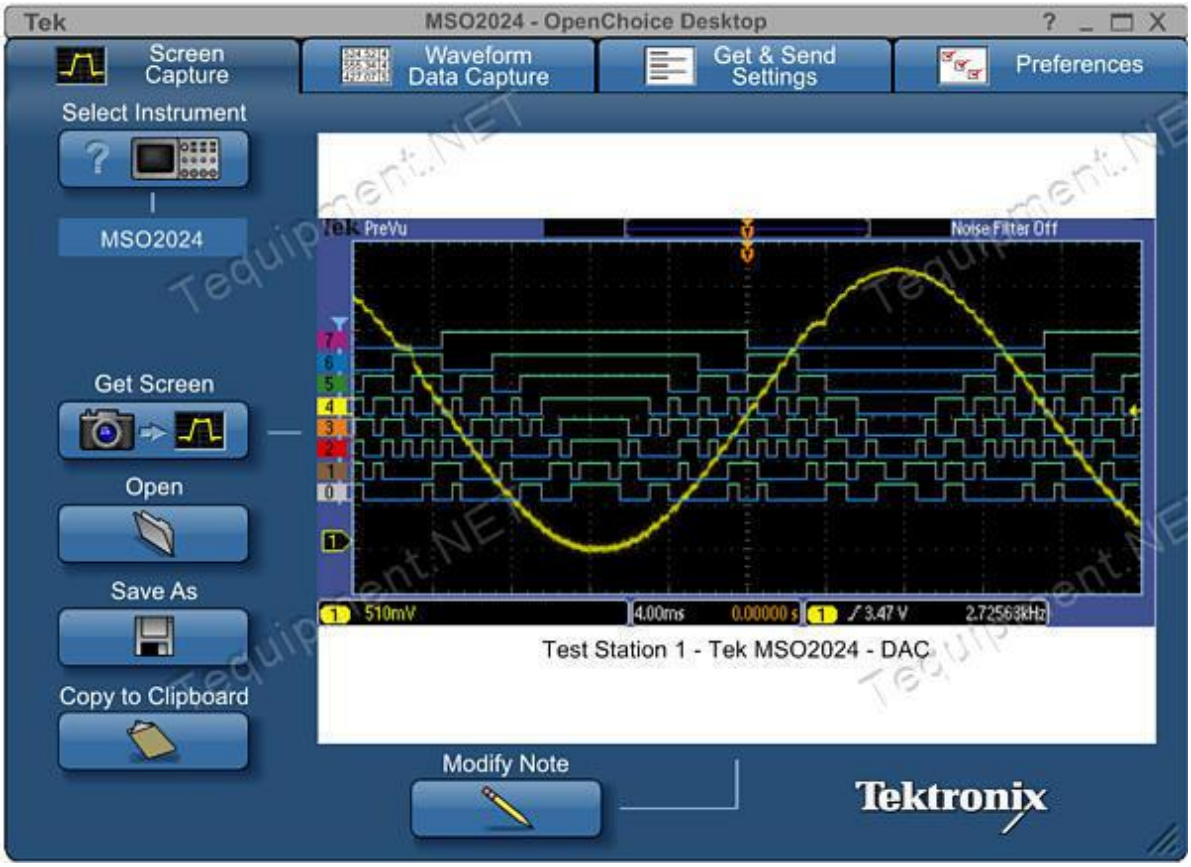

Figure 2: Openchoice Desktop screenshot.

2.2 Approach and decision about alternative solution

Upon seeing that the currently existing alternatives do not reach the ideal, I decided to create my very own software. The alternatives do not reach thr requiremnts because both are exclusive to its own company's hardware, limiting the purchase options of the laboratory, are paid which does not help keep the costs to the minimum and have many superfluous elements that simply complicate the use of the programs whren what is looked for is to keep the program simple and on point to immediately be of help during experimentation without the need to study it to reach the helpful phase.

The new software would be based on the programming language Python and it would connect to the oscilloscopes through USB. A graphical interface would be needed to see the data given by the oscilloscope and some kind of simple User Interface (UI) would be desirable to make the program easy to use.

Python is "an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed." (What is Pyhton? Executive summary https://www.python.org/doc/essays/blurb/)

This programming language is used due to its popularity, meaning users will probably at least have heard of it, ease of installation and the number of libraries, both standard and not, that can be helpful in order to create a program.

With this in mind, I was offered an opportunity to help on a currently ongoing experiment. The Physics Department was doing an experiment to measure the transitory current caused by the shot of a laser onto a sample, measured with an

oscilloscope. The software could help in managing said oscilloscope through configurating its settings and receiving and saving the data obtained.

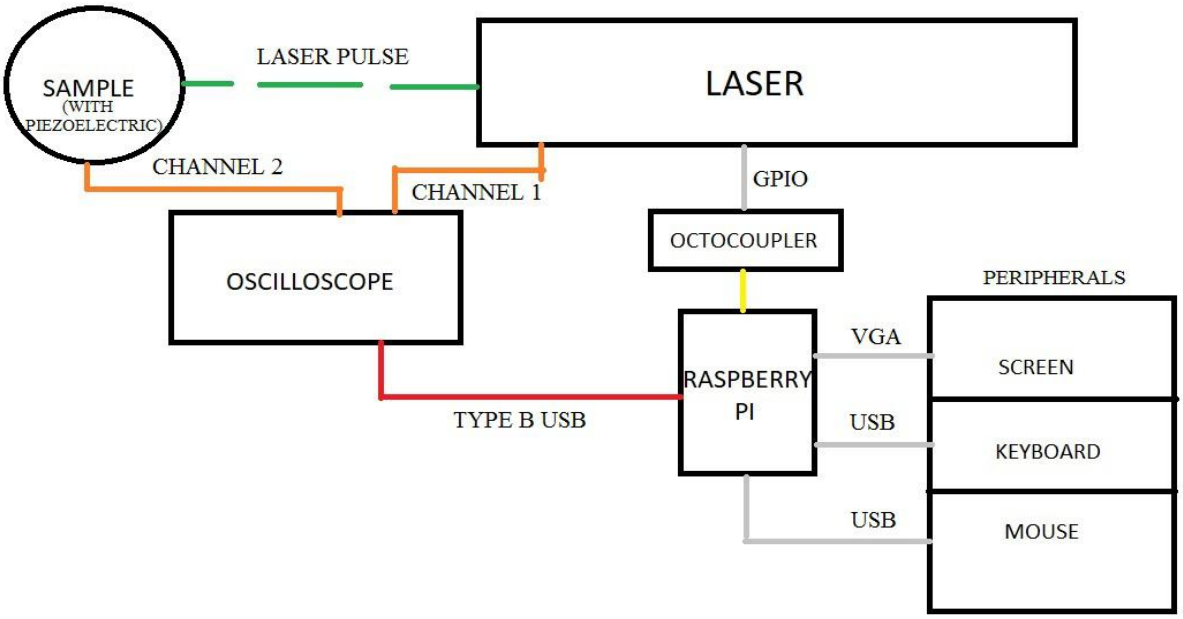

Figure 3: Schematic of the experiment.

The experiment consists of a laser, who is ordered by a computer through a GPIO port, a Raspberry Pi in this case, to shoot against a sample. The shooting signal is a TTL signal and to avoid trouble with it, an octocoupler was added between the Raspberry Pi and le laser. The resulting signal is sent to an oscilloscope through the channel 2 as well as a signal that acts as warning and trigger to the channel 1. This oscilloscope is connected to the same computer that sent the signal by a type B USB cable to save the data for posterior treatment. The computer operation is done through a set of monitor, keyboard and mouse.

However access to the Optics Laboratory is restricted and the equipment used (mainly the laser but also the oscilloscope, a Tektronix DPO 3k series) is expensive, so I could not freely go and come to test my program. That is why, before jumping onto the experiment I designed and created a simulation circuit, a circuit that would send a signal and after a delay send a different one, imitating the behaviour of the laser, that after being given the order to shoot it sends a signal as a trigger to alert the

oscilloscope of the incoming shot and then a second signal to recreate the signal received by the sample shortly after the trigger has been sent.

To do this simulator circuit the needed components would be circuitry to transform a signal into two and to give them a delay, two signal exits to recreate the trigger and the sought signal and the connections for power supply needed which would also work as the initial signal for simplicity's sake. The exit signals would go to an oscilloscope which would be controlled by a computer through USB.

2.3 Development of chosen solution

2.3.1 Design of the simulator circuit

The very first step was the design of the simulation circuit. The necessary components were three integrated circuits, 2 power supply connections (for 3,3V and 5V) and another connections for the grounding, 2 connections for the oscilloscope (the trigger and the signal), 4 resistors (one of 100, one of 5k and two of 10kΩ) 1 regulable resistor (from 100 to 5kΩ, but it was left static at 2kΩ), 2 capacitors (of 0,01 and 470μF) and cable to connect it all.

The three chosen integrated circuits were, in order of installation to the simulator circuit, a 4n25, a LM 324 and a LM 555. The 4n25 is Vishay Semiconductors product that reproduces the octocoupler and supplies the 3,3V and 5V power supplies to the other integrated circuits. The LM 324 and the LM 555 are both Texas Instruments products. The first is an array of 4 operational amplifiers, one of them was used as a current follower directly connected to the oscilloscope, this one being the trigger, while another operational amplifier was used as a current inverter due to the LM 555 delay configuration needing an inverted signal. This LM 555 was as previously specified set up in a configuration that gave a certain delay based on the resistor and capacitors used. All of these components would be placed on a bakelite circuit board and connected with cable using the wire-wrap technique and tin welding. The wire-wrap technique is an interesting point to talk about. It is a manual technique (although it can be automatised) to connect electric contact points in a circuit. The manual variant used in this case uses a special tool, a kind of pen with holes to put the cables, to, as its name indicates, wrap a cable around the wire of the electrical component. The technique was developed in the 60s from experiences obtained on suspension bridges wire ends and telegraph line installation. While it has fallen into disuse, it is still a good option for small prototypes like in our case due to the easiness of use, the only thing needed to use it is to place the components on the board to be used and then placing correctly the wire and the cable on the special pen-like tool and twisting. It is also more mechanically resistant than impressed boards and the

soldering adds even more mechanical resistance and a degree of protection against rust.

The pin to pin configuration of the whole circuit was this one:

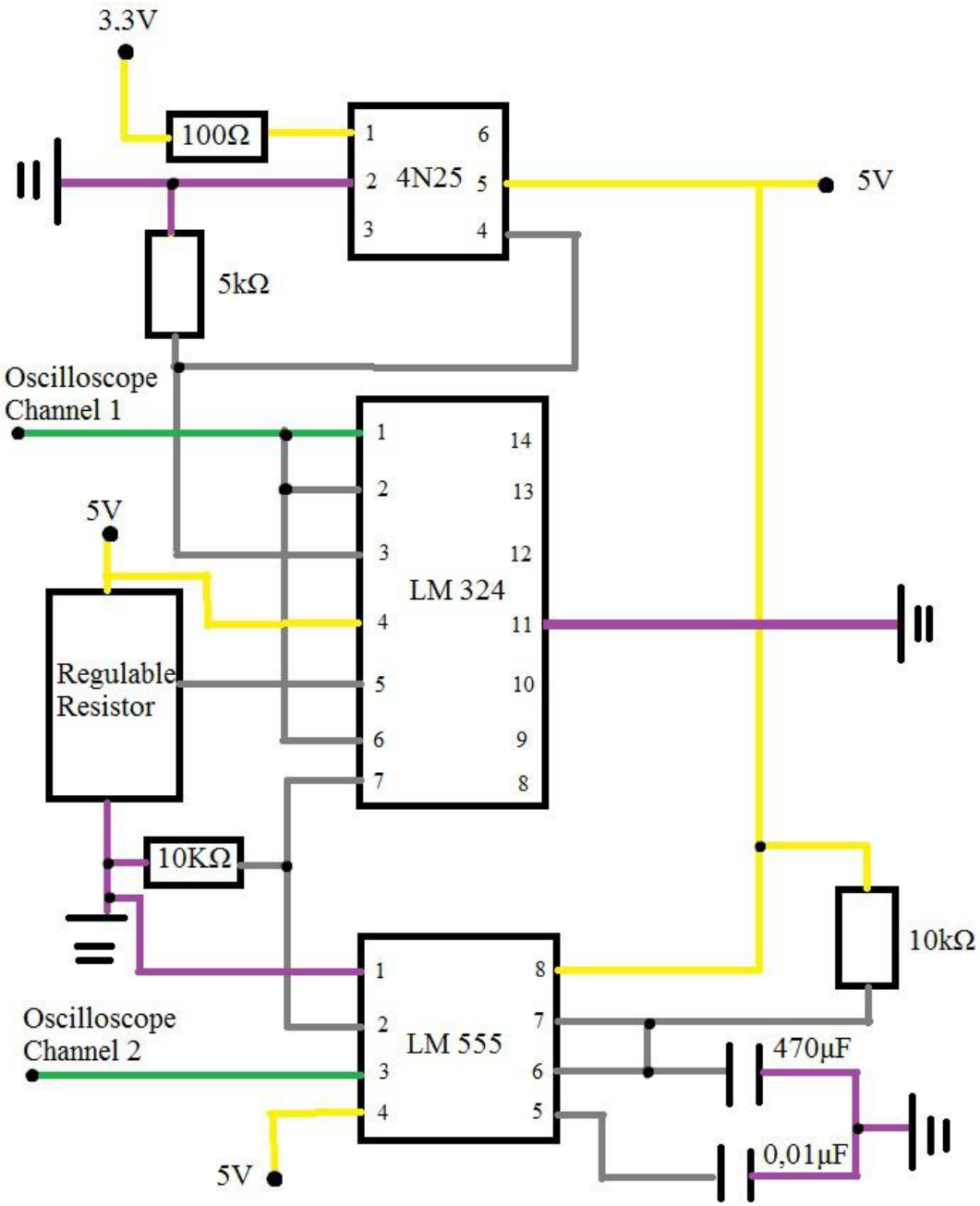

Figure 4: Schematic and pin connections of the simulator circuit.

And the actual circuit would look like this:

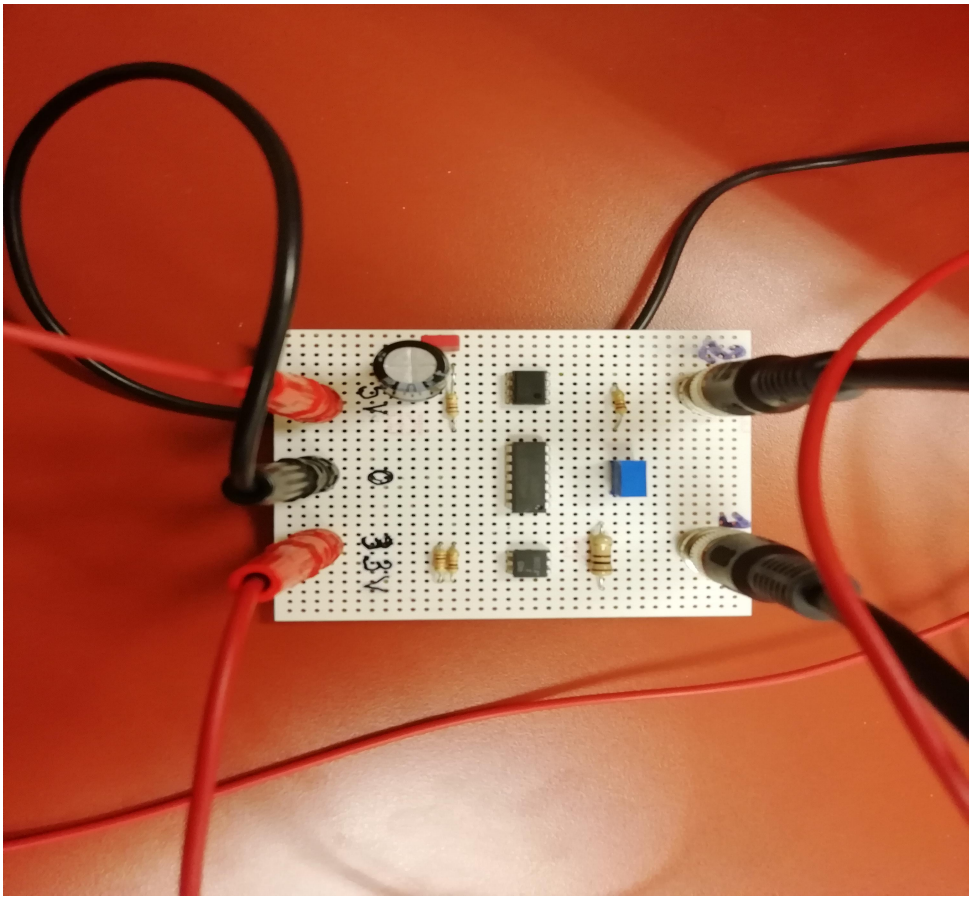

Figure 5: Simulator circuit (upper view)

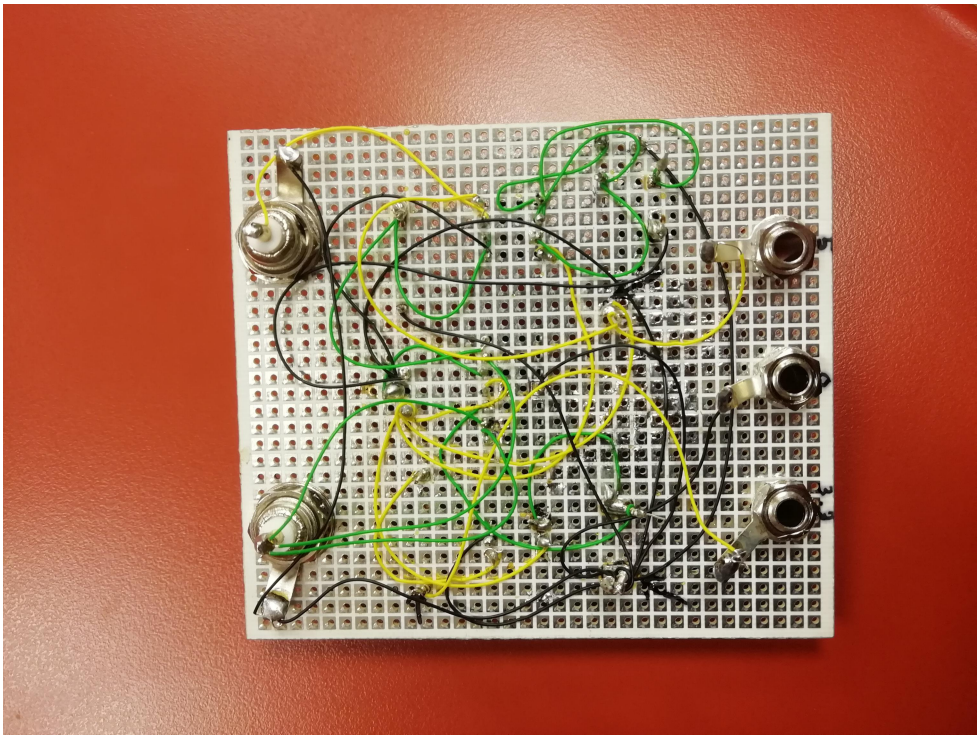

Figure 6: Simulator circuit (under view)

To test it a dual independent power supplier was used and for the oscilloscope the plentiful in teaching laboratories and much cheaper Tektronix TBS1052B was used. The power supplier was set up so that it would provide 3,3V and 5V while the trigger signal was connected to the channel 1 and the delayed signal to the channel 2. Then the oscilloscope was set up by adjusting the vertical and horizontal scales with the appropriate wheels (the ones close to the 1 and 2 of the channel for the vertical scale and the big one at the right of the oscilloscope for the horizontal one) and programming the trigger by entering the Trigger Menu by pushing the button named "Menu" in the Trigger section of the oscilloscope, changing its settings to have an edge type trigger, with the channel 1 as its source, rising slope and trigger activation at an arbitrary value between 1 and 2V.

After this setup the last steps needed were pushing the trigger button to make the oscilloscope wait for the trigger signal and to connect the 3,3V cable to create the signals. The 5V is always already connected with the 3,3V marking the timing of the triggers as the circuit is supplied with energy but the octocoupler does not allow the circuit to produce the signal with the 3,3V not connected.

The circuit works like this:

- The 5V supply to the three integrated circuits the energy needed to operate.

- The 3,3V supply marks the starting point of the operation. It immediately encounters a 100Ω resistor to limit the current, acting as protection in case of shortcircuit.

- The signal reaches the 4n25. The signal will go through pin 4 and reach the LM 324 through pin 3. 4n25 also needs to have pin 5 connected to 5V, pin 2 connected to the ground and pins 2 and 4 connected with a 5kΩ resistor between them.

- Then the signal reaches the first of the four operational amplifiers that the LM 324 has (we only use two). The signal reaches through pin 3 and with pin 2 shortcircuited to pin 1 the first operational amplifier behaves as a current

follower. The outputted signal is sent to oscilloscope's channel 1 to act as trigger but it is also sent to pin 6, towards the second operational amplifier.

- The second operational amplifier acts a a current inverter. The signal reaches through pin 6 while pin 7 acts as the output and pin 5 is connected to the regulable resistor at 2kΩ (it can be changed to alter the gain of the output signal). The resistor is connected the 5V supply along pin 4 of the LM 324 and to the ground along pin 7, although pin 7 is connected to it with a 10kΩ resistor to protect the integrated circuit, avoiding backflow. Pin 11 meanwhile is directly connected to the ground.

- After this the LM 555 integrated circuit is reached. The signal comes through pin 2 and goes to the oscilloscope's channel 2 through pin 3 with a delay changeable through the resistor and capacitors of the right side pins.

- Pins 4 and 5 are connected to the 5V supply, pin 1 is connected to the ground and pins 6 and 5 are also connected to the ground but through a 470μF and a 0,01μF capacitors respectively. Pin 7 shortcircuits pin 6 and is connected to pin 3 with a 10kΩ resistors. This last resistors and capacitors are the ones that modify the delay.

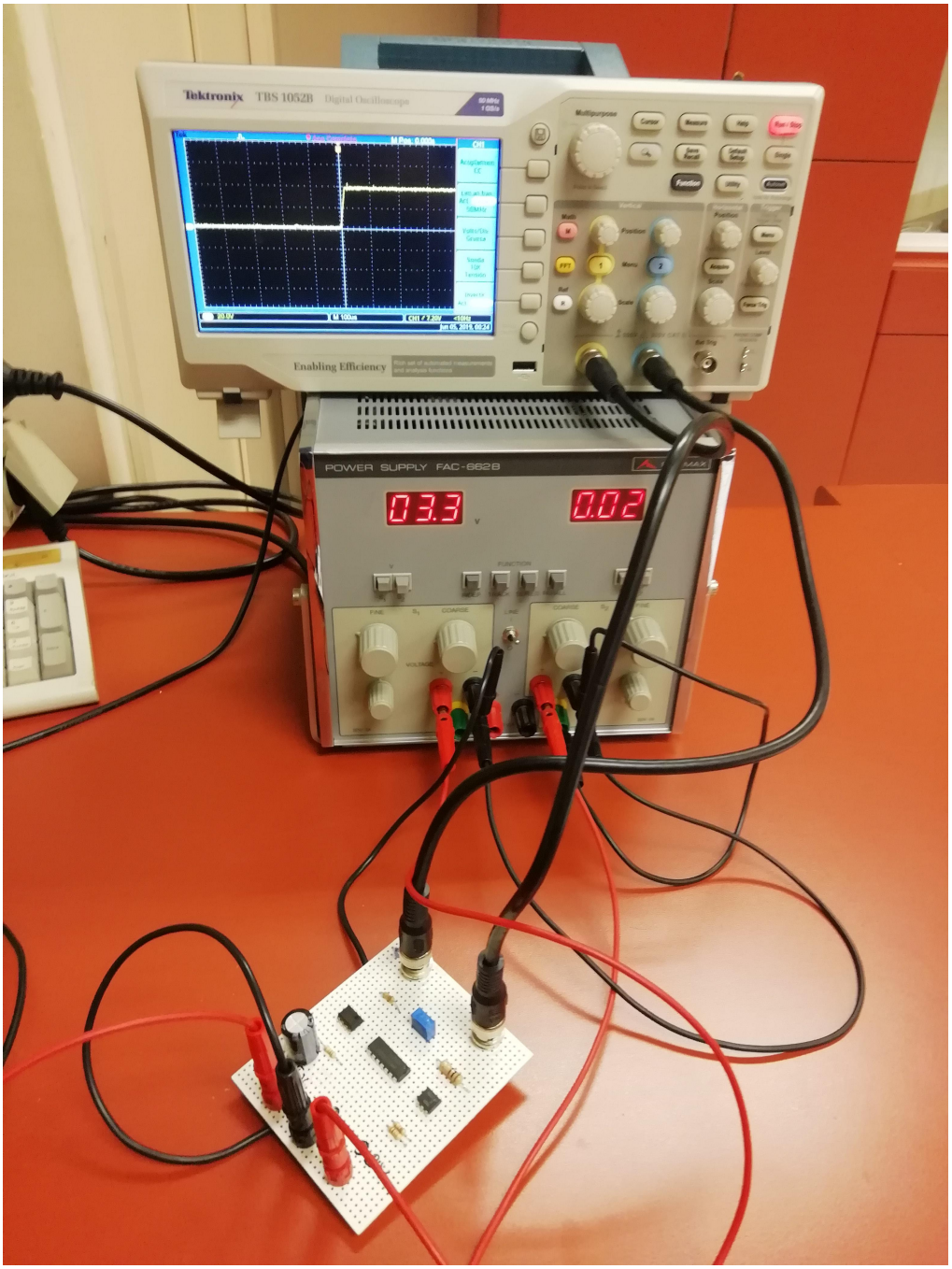

Figure 7: Connected simulator circuit.

2.3.2 Creation of a program to control the oscilloscope

Once the simulator circuit has been created and shown to be operational the next step is to create the software I aim for. To create it I downloaded Python into my personal computer. Python was chosen for being a very useful and worked with programming language with tons of useful libraries that could help in the endeavour. I specifically downloaded Python 3 as Python 2 is in 2020 going to go out of service. To install Python3 we just need to introduce in the terminal:

>>>sudo apt install python3.6

It must be taken into consideration that this commands and the rest of Terminal commands used in this report are Ubuntu based ones because this project was done with a computer that uses Ubuntu as its operative system. Those libraries are the solution for our next problem. Controlling the oscilloscope is not as easy as connecting the oscilloscope with a USB and writing Python code to order it around. The first problem is the connection to the oscilloscope. Just connecting them with a Type B USB cable is not enough as the computer and the oscilloscope cannot communicate with each other as they have no drivers installed to do so, in other words, they are in the same room but forget the language, they do not even know how to speak with each other. The tool used to make them learn how to communicate with each other is USBTMC. USBTMC is a protocol that allows communication of hardware through USB. In this case we want to give Python this ability so a library created for this was downloaded through ”pip”:

>>>pip install python-usbtmc

But before this it might be interesting to look at virtual environments. As explained, a number of libraries are needed and so, due to Python being the programming language used, a Python virtual environment was created. A virtual environment is a tool to keep dependencies and libraries independent of other projects and of the main system. The creation of one can fully be done through the terminal. First we install the “venv” tool with pip:

```
>>>pip install virtualenv
```

Then a new directory is created and after travelling with the command “cd” to it the command to create the virtual environment is given:

```
>>>mkdir python-virtual-environments && cd python-virtual-environments

>>>python3 -m venv env
```

Afterwards, whenever we want to use this virtual environment the only thing that needs to be done is travel until the bin folder inside the directory of the created virtual environment and give the activate command:

```
>>>source activate
```

The virtual environment protect the main system by separating it from the work environment and simplifies the task of listing and checking the libraries used by the program.

Back again onto the project, once the oscilloscope and the computer know how to speak with each other thanks to PyUSB and USBTMC, they need a common language. Having used Python libraries, Python is obviously used but it is not enough. Tektronix oscilloscopes, and virtually all oscilloscopes, use a certain language to receive and return queries, SCPI. SCPI came to be in the 90s as a standard solution to the problem that while connection with measurement equipment was standardised through IEEE-488 buses, sending commands was not so. USBTMC allows the use of Python instructions in which SCPI language can be introduced. With all of these together, the problem of how to communicate a computer with a oscilloscope is resolved.

Once this setup is complete the programming enters the spotlight. The very first thing that needs to be done when writing the .py file used to command the oscilloscope is to import the usbtmc library. Afterwards another always necessary step is to create a variable that will embody the oscilloscope. The difficult part of it is that to do so the ID of the oscilloscope is needed and it changes based on model. To find out the ID of the oscilloscope being used is to connect the oscilloscope with the USB cable and

then ask the computer about the devices connected to it via USB. In Ubuntu it is as easy as using the command “ls -usb” and a list of devices should appear with one of them being something similar to “Tektronix Oscilloscope (699:368)” (this concrete ID is the one of the TBS1052 oscilloscope used during most of this TFG experimental phase). The looked for ID are the numbers. They have to be inputted with a 0x because these numbers are actually hexadecimal ones. To finish the first lines of the programs the communication channel must be opened, which unsurprisingly is done with the command “open”. And so, the first lines of our program would look like this:

```
import usbtmc
instr = usbtmc.Instrument (0x699, 0x368)
instr.open()
```

This is an example for communication with the TBS1052 oscilloscope but if we wanted to communicate with another oscilloscope the number “699” and “368” would need to be changed to the matching one of the oscilloscope we want to use. The 0x does not disappear in any case, it is necessary for the program to understand the command. Because it would be good to be prepared to do so, a method to know the model used is also run at the beginning of the program. In this program there is no difference between using one or the other but if both oscilloscopes run the code without error we know that the structure created can be brought to the final program. However, this still has the flaw that the numerical ID must still be manually written onto the code, a way for the program to automatically check the ID has not been implemented.

With the base established, my next objective was to create a program that would save the initial settings of the oscilloscope, changing them to the ones I wished to use, giving a warning and some time to check if the settings have been properly modified and finally restore the initial settings saved at the beginning of the program’s execution. This program can be found in Annex 1: save_launch_load.py and I will proceed to explain it here.

```
import usbtmc
import time
```

```
instr = usbtmc.Instrument(0x699, 0x368)
scopename = instr.ask("*IDN?")
scopename = scopename.split(',')
model = scopename[1]
instr.open()
prevtig=False
```

As previously showed the first lines of code correspond to the exportation of the USBTMC library, the naming and creation of a variable for the oscilloscope and the opening of the communication channel. However there is also the exportation of another library, the installed by default with python “time” library. This library is needed because the TBS1k series is a series launched in 2012, its processing power is quite low and giving many orders can make him skip some of them. That’s why we used the command “sleep” to create delays, giving time to the oscilloscope to execute all the commands given. Also, as stated, an array and split that take into account the answer to “(“*IDN?”)” are used to recognise the model.

After this we enter the first section of the program, saving the settings that will be changed during the second section. I decided to only save the settings that would be interesting to be able to change in this simulation and in the laser experiments because no other settings would be touched so there would be no reason to change them and because finding all the settings of the oscilloscope would take a very big amount of time. The very first line of the save section is the creation of a variable that will be used in case the initial trigger is an edge time and it begins as False. This variable is used in both the first section and the third section and exists to avoid manipulating edge trigger settings that are exclusive to said kind of trigger, that way the user would not give orders that the oscilloscope could not process, those orders being saving and loading settings that the oscilloscope currently has no access to (for example, saving the edge’s slope setting when video type trigger has no slope setting). The oscilloscope seemingly ignores these kind of commands so maybe it is not a necessary measure but better safe than sorry. After this initial precaution a methodical process is followed. The setting to be saved gets its own variable and in its very call a query for the oscilloscope to give the setting is launched:

```
#Saving altered variables

trigg=instr.ask("TRIGger:MAIn:TYPe?")
triggch=instr.ask("TRIGger:MAIn:EDGE:SOUrce?")
trigglev=instr.ask("TRIGger:MAIn:LEVel?")
if trigg=="EDGE":
        prevtig=True
        triggslope=instr.ask("TRIGger:MAIn:EDGE:SLOpe?")
        triggcoupl=instr.ask("TRIGger:MAIn:EDGE:COUPLing?")

time.sleep(0.5)

dispch1=instr.ask("SELect:CH1?")
dispch2=instr.ask("SELect:CH2?")

time.sleep(0.5)

ch1coupl=instr.ask("CH1:COUPLing?")
ch1probe=instr.ask("CH1:PRObe?")
ch2coupl=instr.ask("CH2:COUPLing?")
ch2probe=instr.ask("CH2:PRObe?")

time.sleep(0.5)

ych1=instr.ask("CH1:VOLts?")
ych2=instr.ask("CH2:VOLts?")

time.sleep(0.5)

xaxistime=instr.ask("HORizontal:MAIn:SCAle?")

time.sleep(0.5)
```

```
triggstat=instr.ask("ACQuire:STOPAfter?")
startstat=instr.ask("ACQuire:STATE?")

time.sleep(0.5)
```

As an example I will explain the very first saved setting. “trigg” is the name given to the variable that will house the initial state of type of trigger. “instr” is the name we have given to the oscilloscope, which acts kind of like a class inside the program. ask is the usual command used by USBTMC to query the oscilloscope. “TRIGger:MAIn:TYPe?” is the SCPI language way of asking the oscilloscope about its current type of trigger. As you can see it needs to be in a string format and the ? is the query indicator in SCPI. This process is repeated for all the settings of interest with the only exception of the exclusive edge type trigger settings trigger slope and trigger coupling which are only saved if the initial trigger is edge type which is checked by comparing our trigger type variable “trigg” with the answer that would be given by the oscilloscope if the trigger type was edge, EDGE. This flags “prevtig” as True, which will be checked in the third section, the load section, to let the program know if it needs to loads the edge exclusive settings or if it is not necessary. As it may have been noticed, SCPI has a curious use of capital letters. These command are stated to be written like these by the Tektronix programming manuals checked but it seems there can be a bit of leeway on not being absolutely correct on its capitalization syntax, however writing them properly assures finding no syntax errors afterwards.

The next part is the second section, the launch section. It guides the program into a certain route depending on the model it recognises because some orders can be written differently or some values might not be appropriated to all oscilloscopes. At this stage it is not that important but it will be relevant on the final program at the time of acquiring the data for the plot, so having a guide structure that works is a help for that moment to come. This section has no variable and simply uses the write function in the USBTMC library to send the changes to the oscilloscope:

```
#Begin setup

if    'TBS' in model:
```

```
        instr.open()
        instr.ask("*ESR?")
        instr.write("*CLS")
        instr.write("ACQUIRE:STATE STOP")
        instr.write("ACQUIRE:STOPAFTER SEQUENCE")
        instr.write("ACQUIRE:STATE RUN")
        instr.write("DATa:SOUrce CH1")
        print("Waiting for trigger...")
        print(instr.ask("BUSY?"))
        while instr.ask("BUSY?")=="1":
            pass
        instr.ask("*OPC?")
        print("Trigger received")

elif 'DPO' in model:
        instr.open()
        instr.ask("*ESR?")
        instr.write("*CLS")
        instr.write("ACQUIRE:STATE STOP")
        instr.write("ACQUIRE:STOPAFTER SEQUENCE")
        instr.write("ACQUIRE:STATE RUN")
        instr.write("DATa:SOUrce CH1")
        print("Waiting for trigger...")
        print(instr.ask("BUSY?"))
        while instr.ask("BUSY?")=="1":
            pass
        instr.ask("*OPC?")
        print("Trigger received")
```

As before, every time we communicate with the oscilloscope we do it through “instr” and this time, as it is not a query but command sent from the computer, we use “write” instead of “ask”. This time instead of a “?” we add a space and write the value we want modified. Whether that value is letter (DC) or numbers (5.0E-5) the need to be strings. Another curious factor about numeric values is that the oscilloscope will go

for the closest value if the value sent is not one it has (for example, DPO3k series oscilloscope have no 5.0E-6 value in the horizontal scale, so writing it will have the oscilloscope change to the closest value, 4.0E-6).

Finally the third sections is reached, the load section. This section uses the variables created on the first section to give the changed variables the values they had before they were changed:

```
#Load initial saved settings

instr.write("TRIGger:MAIn:TYPe %s" %(trigg))
instr.write("TRIGger:MAIn:EDGE:SOUrce %s" %(triggch))
instr.write("TRIGger:MAIn:LEVel %s" %(trigglev))
if prevtig==True:
    instr.write("TRIGger:MAIn:EDGE:SLOpe %s" %(triggslope))
    instr.write("TRIGger:MAIn:EDGE:COUPLing %s" %(triggcoupl))

time.sleep(0.5)

instr.write("SELect:CH1 %s" %(dispch1))
instr.write("SELect:CH2 %s" %(dispch2))

time.sleep(0.5)

instr.write("CH1:COUPLing %s" %(ch1coupl))
instr.write("CH1:PRObe %s" %(ch1probe))
instr.write("CH2:COUPLing %s" %(ch2coupl))
instr.write("CH2:PRObe %s" %(ch2probe))

time.sleep(0.5)

instr.write("CH1:VOLts %s" %(ych1))
instr.write("CH2:VOLts %s" %(ych2))
```

```
time.sleep(0.5)

instr.write("HORizontal:MAIn:SCAle %s" %(xaxistime))

time.sleep(0.5)

instr.write("ACQuire:STOPAfter %s" %(triggstat))
instr.write("ACQuire:STATE %s" %(startstat))
```

As it modifies a value inside the oscilloscope “write” is the used command. Where in the second section the wished value would go, in this section “%s” is used to mark that the string of a variable is used. As stated before, even if they are numbers (in this case it is a 1) the values need to be strings for the oscilloscope to be able to read them. A check for the initial trigger type through an “if” is the only thing that breaks the monotony of this section’s code lines. A “instr.close()” order is not necessary to finish the program.

With this program done I showed the ability to save change and load settings of an oscilloscope through a computer. The following step was to create a proper program instead of a one run series of code lines.

2.3.3 Creation of the objective software program with a WxGlade interface

The easiest way to get a framework for the program with a graphical interface is to use WxGlade, or at least it is the option I went for. WxGlade is great to make simple to the point UIs and it is compatible with the commonly used Operative Systems (OSs) and with the programming language used, Python 3. To begin using it the only thing that needs to be done is to download it through the terminal or through its sourceforge web page. Once downloaded the only thing needed to start working with it is to use the command “wxglade” on the terminal:

```
>>>sudo apt-get install wxglade
>>>wxglade
```

However before beginning to work with WxGlade knowing what you want to do is a great help. What I wanted was a program with buttons for the different settings to be changed so that when clicked would ask the user to input the value they want that setting to change, then a space to visualize the data extracted from the oscilloscope and finally a way to save the extracted data. With this in mind I thought the simplest way to do it would be to make a menu bar in the typical File/Edit/View/ Tools/Help style with the different buttons to manipulate the program inside so as to not occupy to much space, that would be used to show two grids that would display the data acquired by the oscilloscope.

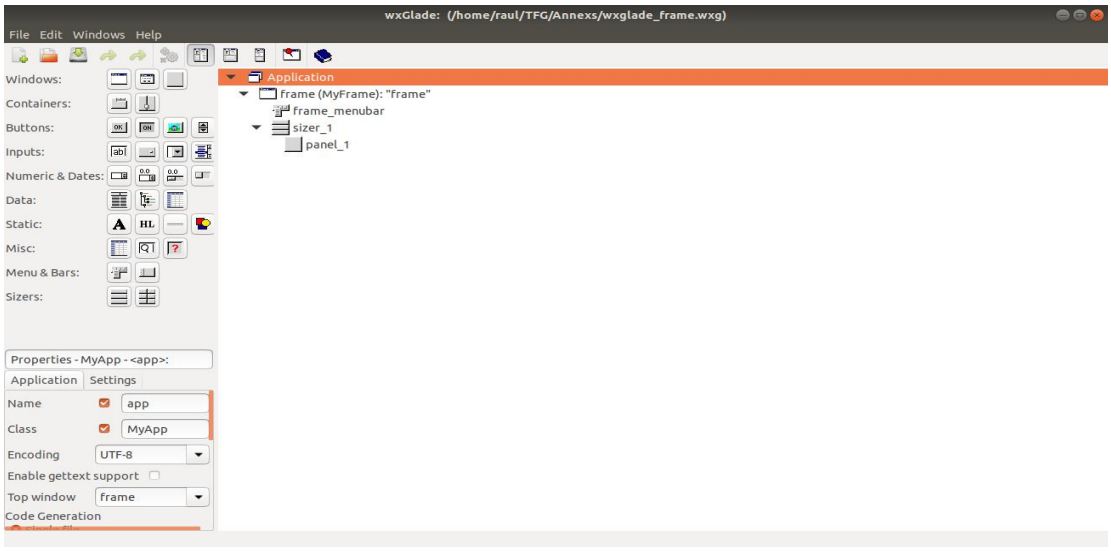

Figure 8: WxGlade main graphical interface.

The menu bar creation is simple as WxGlade already has an option dedicated to it. The only work that needs to be done is creating the different collapsible menu options and then adding to them the buttons we wanted to use, like opening a certain data file, changing the vertical axis of one channel, etc. And so the very first step in using WxGlade is to use the option “Add a MenuBar”. This allows to create collapsibles (the non-indented) with the buttons I want to send certain commands (the indented). This page also gives every added event its own event handler that will be the reference used when programming. In this menu bar the File collapsible has the options for file management (open a file, save it, save it as) and the option to restore the initial settings of the oscilloscope, from the point the program has been initialized.

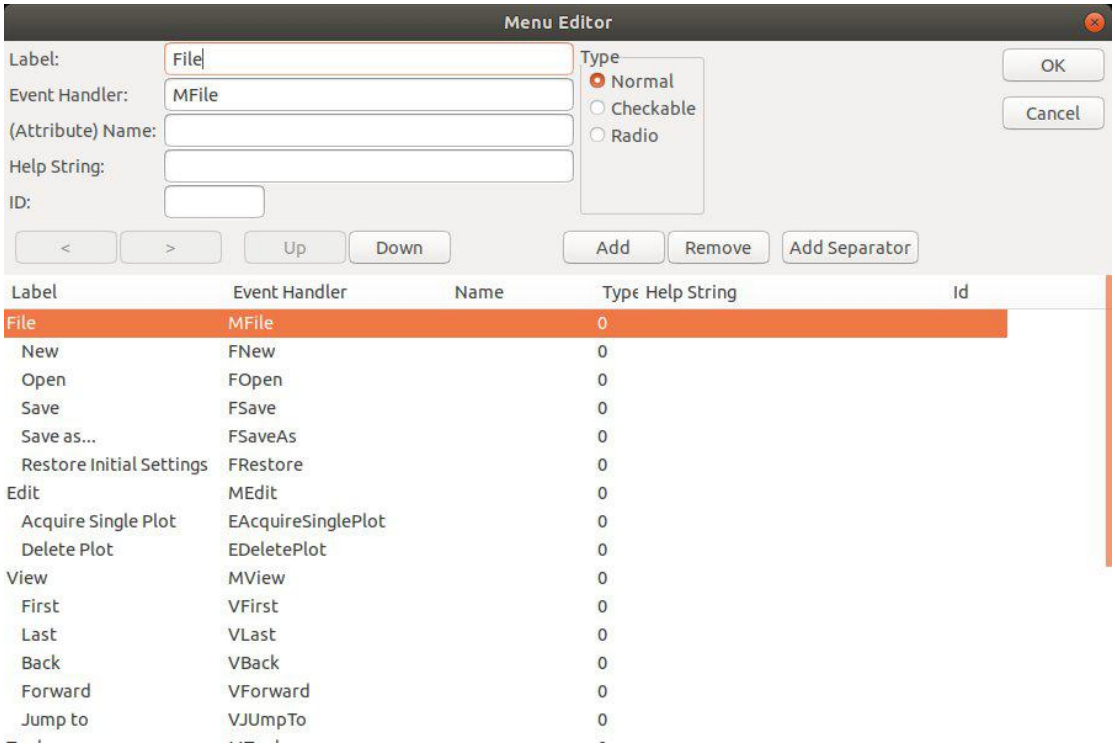

Figure 9: WxGlade Menu Editor.

Afterwards a “sizer” is added which inside has a “panel” and with that we already have a zone to display the extracted data. Now to start programming we need where to write our program. We can generate a file by clicking on “Application” and at the bottom of the “Widget” tab we find “Generate Source Code”. We generate a python file by writing the name of the file finished with a .py (in this case “wxglade_frame_gui.py”) and this will be the base of the programming effort. However as this file is directly limked to WxGlade and is susceptible to be changed

by it, we create a new Pyhton file (“tfg_program.py”) that will export the generated source file. Both files are available in Annex 2: wxglade_frame_gui.py and Annex 3: tfg_program.py. Annex 2: wxglade_frame_gui.py does not require much explaining, it is the file generated by WxGlade and nothing should be written on it, it is just there to serve as base for the program that will be actually run which is “tfg_program.py”. The only point of note it has is that it is the responsible for the program to remain running, usually Python programs run through its lines once and after executing everything the need to, close themselves. Annex 3: tfg_program.py is where programming retakes the spotlight and I will proceed to explain its composition.

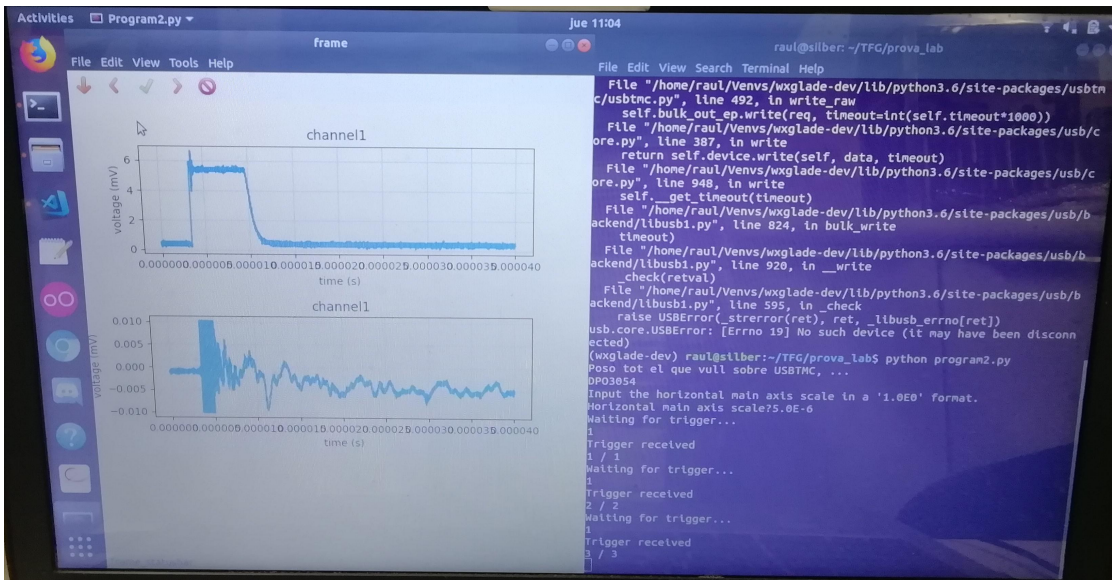

Figure 10: Program frame and Terminal.

Before that, it is interesting to note that while the graphical interface provides the user with quick ways to manipulate the oscilloscope settings and a way to see the extracted data, the terminal is where further instructions are provided while acting as a status bar of sorts. The UI has a vertical distribution so the user can easily use half the screen with the UI and the other half with the terminal.

```
import wx
from struct import unpack
import usbtmc
import time
import numpy as np
```

```
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as +
+ FigureCanvas
from matplotlib.figure import Figure
from wxglade_frame_gui import MyFrame
import pickle as pk
import os
```

The first thing to do is to import everything that will be needed. In no particular order, “usbtmc” is imported to communicate with the oscilloscope, “wx” is used to be able to show the UI, “struct”, “pickle” and “os” are imported for saving and loading data, “numpy” and some “matplotlib” functions are imported to show graphs of the data obtained and from “wxglade_frame_gui” the class “MyFrame” is imported because in it there is all the event handlers (which are basically functions) that activate when pressing the corresponding button and run the code that is programmed within this “tfg_program.py” file.

```
class DataPlot:
    def __init__(self, t, y1, y2,):
        self.t = t
        self.y1 = y1
        self.y2 = y2
        return
```

After all this imports the class “DataPlot” is found. This class exists to contain all the plots obtained by querying the data obtained by the oscilloscope. All plots contained in this class have its data contained in 3 variables: “t” for the time or x axis, “y1” for the curve formed by the data from channel 1 and “y2” for the curve formed by the data from channel 2. The function that describes this is named “__init__” because functions with that name are run at the beginning of the program, when it initializes. We do so in this case because we want to be able to work with the data from the start, be it to acquire new one or load old one.

```
class DataSet:
    def __init__(self):
```

```
        self.plots = []
        self.display = -1
        return

    def append(self, t, y1, y2):
        plot = DataPlot(t, y1, y2)
        self.plots.append(plot)
        self.display = len(self.plots) - 1
        self.print_position()
        return

    def remove(self):
        if self.display < 0:
            return

        self.plots = self.plots[:self.display]+self.plots[self.display+1:]
        if self.display == len(self.plots):
            self.display = self.display - 1
        self.print_position()
        return

    def forward(self):
        if self.display < 0:
            return

        if self.display < len(self.plots) - 1:
            self.display = self.display + 1
        self.print_position()
        return

    def backward(self):
        if self.display > 0:
            self.display = self.display - 1
        self.print_position()
```

```
        return

    def goto(self, plot_index):
        if self.display < 0:
            return
        plot_index = int(input("Introduce the postion of the plot:"))
        if plot_index > 0 and plot_index < len(self.plots) + 1:
            self.display = plot_index - 1
        self.print_position()
        return

    def print_position(self):
        print(self.display + 1, "/", len(self.plots))
        return

    def display_data(self, frame):
        if self.display < 0:
            return
        X = self.plots[self.display].t
        Y = self.plots[self.display].y1
        Z = self.plots[self.display].y2
        frame.ax1.cla()
        frame.ax2.cla()
        frame.ax1.plot(X, Y)
        frame.ax1.set(xlabel='time (s)', ylabel='voltage (V)', title='channel1')
        frame.ax1.grid()
        frame.ax2.plot(X, Z)
        frame.ax2.set(xlabel='time (s)', ylabel='voltage (V)', title='channel2')
        frame.ax2.grid()
        frame.canvas.draw()
        return
```

Afterwards comes the “DataSet” class. This class contains the functions to do the data treatment and display of it. It starts with a “__init__” function that creates the “plots”

array, an array that will contain the different sets of data that have been extracted from the oscilloscope, and setting the display to show nothing and give a starter point which will be useful when acquiring data. The “append” function introduces a set of data into “plots”, gives the order to display this last plot of obtained data and orders to display the new position of the viewed plot (this last part will be explained in the “print_position” function). The “remove” function deletes the currently viewed plot and as with the rest of “DataSet” functions returns the user the current position after the operation. The “forward” and “backward” move the display to the next or previous plot respectively. They are written so that “forward” gives no problem when asking to do so in the last plot and the same with the first plot and “backward”. The function “goto” asks the user to which plot position wishes to jump and after checking that the position introduced is a valid one it displays said position. It bears mention saying that the user does not have to counter intuitively count from 0 even though the array actually has the first plot in the 0 position, the function already corrects it for him. Also, the five last functions are explicitly written to do nothing when there is not a single plot extracted to avoid any errors. The following function is “print_position”. This function acts as guide for the user to tell them which plot are they visualizing from the currently opened data set and how many plots in total are there in the set. Just as the “goto” function, “print_position” shows the first plot of the set as the plot number 1 to avoid confusing the user. The last function in the “DataSet” class is “display_data”. This function used the data extracted from the oscilloscope to display it in the graphical interface in the form of 2 graphed plots, one for each channel. It takes the horizontal information as “X” to from the shared between the two plots x axis, the curve from the channel 1 as “Y” and the curve from channel 2 as “Z”. Then it clears the currently displayed plots in case there was already a plot shown, creates the new plots, gives arbitrary labels to the axis and a title to the graphs, prepares a grid for them and finally shows them in the screen. Like with “print_position” this function has calls to it every time a function that produces a change in the data is run (opening a saved data set, removing a plot, forwarding or jumping,...). ax1 and ax2 are the variable names given to the upper and lower graphs.

```
class ElementalFrame(MyFrame):
    def __init__(self, *args, **kwds):
        MyFrame.__init__(self, *args, **kwds)
```

```
print("Welcome.")
self.figure = Figure()
self.ax1 = self.figure.add_subplot(211)
self.ax2 = self.figure.add_subplot(212,sharex=self.ax1)
self.figure.subplots_adjust(hspace=0.6, wspace=0.4)
self.canvas = FigureCanvas(self.panel_1, -1, self.figure)
self.data_set = DataSet()
self.data_path = ""

self.instr = usbtmc.Instrument(0x699, 0x415) #DPO 3054
#self.instr = usbtmc.Instrument(0x699, 0x368) #TBS 1052B
self.scopename = self.instr.ask("*IDN?")
self.scopename = self.scopename.split(',')
self.model = self.scopename[1]
self.instr.timeout = 15
self.instr.open()
self.trigg=self.instr.ask("TRIGger:MAIn:TYPe?")
self.triggch=self.instr.ask("TRIGger:MAIn:EDGE:SOUrce?")
self.trigglev=self.instr.ask("TRIGger:MAIn:LEVel?")
if self.trigg=="EDGE":
    self.prevtig=True
    self.triggslope=self.instr.ask("TRIGger:MAIn:EDGE:SLOpe?")
    self.triggcoupl=self.instr.ask("TRIGger:MAIn:EDGE:COUPLing?")

time.sleep(0.5)

self.dispch1=self.instr.ask("SELect:CH1?")
self.dispch2=self.instr.ask("SELect:CH2?")

time.sleep(0.5)

self.ch1coupl=self.instr.ask("CH1:COUPLing?")
self.ch1probe=self.instr.ask("CH1:PRObe?")
self.ch2coupl=self.instr.ask("CH2:COUPLing?")
```

```
        self.ch2probe=self.instr.ask("CH2:PRObe?")

        time.sleep(0.5)

        self.ych1=self.instr.ask("CH1:VOLts?")
        self.ych2=self.instr.ask("CH2:VOLts?")

        time.sleep(0.5)

        self.xaxistime=self.instr.ask("HORizontal:MAIn:SCAle?")

        time.sleep(0.5)

        self.triggstat=self.instr.ask("ACQuire:STOPAfter?")
        self.startstat=self.instr.ask("ACQuire:STATE?")

        time.sleep(0.5)
        return
```

Finished with the two classes created by the user we enter into the classes created by program sourced by WxGlade. The class “ElementalFrame” has all the event handlers that have been created when making the menu bar. Moreover, as already usual, it has an “__init__” that creates the graphical interface, the two graphs, creates a default data set to start working and gives a blank save path to avoid overwriting data sets by accident. As in the previous annex, the program needs to be edited to change the value of “usbtmc.Instrument()” to the match the used oscilloscope in order to be able to communicate with it. It also has the save section of Annex 1: save_launch_load.py to store the settings of the oscilloscope at the time of initialization.

```
    def FNew(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.__init__()
        self.data_path = ""
        self.ax1.cla()
```

```
            self.ax2.cla()
            self.canvas.draw()
            return

        def FOpen(self, event):    # wxGlade: MyFrame.<event_handler>
            wildcard = "Pickle file (*.pkl)|*.pkl|" "All files (*.*)|*.*"
            dlg = wx.FileDialog(self, message="Choose a file", defaultDir=os.getcwd(),
defaultFile="", wildcard=wildcard, style=wx.FD_OPEN)
            if dlg.ShowModal() == wx.ID_OK:
                paths = dlg.GetPaths()
                print("You chose the following file(s):")
                for path in paths:
                    print(path)
                self.data_path = paths[0]
            dlg.Destroy()
            inputfile = open(self.data_path, 'rb')
            self.data_set = pk.load(inputfile)
            inputfile.close()
            self.data_set.display_data(self)
            return

        def FSave(self, event):    # wxGlade: MyFrame.<event_handler>
            if self.data_path == "":
                self.FSaveAs(event)
            else:
                outputfile = open(self.data_path, 'wb')
                pk.dump(self.data_set, outputfile)
                outputfile.close()
            return

        def FSaveAs(self, event):    # wxGlade: MyFrame.<event_handler>
            wildcard = "Pickle file (*.pkl)|*.pkl|" "All files (*.*)|*.*"
            dlg = wx.FileDialog(
```

```
            self, message="Save file as ...", defaultDir=os.getcwd(), wildcard=wildcard,
defaultFile="data.pkl", style=wx.FD_SAVE)
            if dlg.ShowModal() == wx.ID_OK:
                self.data_path = dlg.GetPath()
                print("You chose the following filename: %s" % self.data_path)
            dlg.Destroy()
            outputfile = open(self.data_path, 'wb')
            pk.dump(self.data_set, outputfile)
            outputfile.close()
            return

        def FRestore(self, event):    # wxGlade: MyFrame.<event_handler>
            self.instr.write("TRIGger:MAIn:TYPe %s" %(self.trigg))
            self.instr.write("TRIGger:MAIn:EDGE:SOUrce %s" %(self.triggch))
            self.instr.write("TRIGger:MAIn:LEVel %s" %(self.trigglev))
            if self.prevtig==True:
                self.instr.write("TRIGger:MAIn:EDGE:SLOpe %s" %(self.triggslope))

self.instr.write("TRIGger:MAIn:EDGE:COUPLing %s" %(self.triggcoupl))

            time.sleep(0.5)

            self.instr.write("SELect:CH1 %s" %(self.dispch1))
            self.instr.write("SELect:CH2 %s" %(self.dispch2))

            time.sleep(0.5)

            self.instr.write("CH1:COUPLing %s" %(self.ch1coupl))
            self.instr.write("CH1:PRObe %s" %(self.ch1probe))
            self.instr.write("CH2:COUPLing %s" %(self.ch2coupl))
            self.instr.write("CH2:PRObe %s" %(self.ch2probe))

            time.sleep(0.5)
```

```
            self.instr.write("CH1:VOLts %s" %(self.ych1))
            self.instr.write("CH2:VOLts %s" %(self.ych2))

            time.sleep(0.5)

            self.instr.write("HORizontal:MAIn:SCAle %s" %(self.xaxistime))

            time.sleep(0.5)

            self.instr.write("ACQuire:STOPAfter %s" %(self.triggstat))
            self.instr.write("ACQuire:STATE %s" %(self.startstat))
        return
```

After this comes the turns for the functions created by WxGlade's event handlers. While the 5 collapsibles have functions because an event handler is obligatory when adding parts to the menu bar, we leave them as "event.Skip()" which is their default state because I do not want them to do anything but show the options that are kept inside them and that does not need extra programming, it is already done on the gui part. The first event handlers are the ones that belong to the File collapsible. "FNew" basically restarts the program data wise by reinitializing the user created class "DataSet", that way we can create a new data set without having to close and open the program. "FOpen" is better explained alongside "FSaveAs". "FSaveAs" is the reason why "os", "pickle" and "unpack" are imported. This function opens a dialogue box to ask the location and name of the data set to save. This data can be saved anywhere but only in .pkl format because it is the simplest way to do so and remembers the file path so that "Save" commands can be used to save the data sets and overwrite them. The "FOpen" opens a dialogue box that lets the user choose the data set in .pkl format they want to upload to the graphical interface in case they want to add, remove or check the plots inside it. "FSave" acts differently depending if the current data set has been saved already or not. If the file path is a blank string it call the "FSaveAs" function, if not it overwrites the old data in the path file for the present data. Then "FRestore" contains the third part of Annex's 1: save_launch_load.py which writes back the settings the oscilloscope gave to the program when it was initialized. As

stated before, the settings it writes are the settings the program automatically acquires when initializing itself.

```
    def EAcquireSinglePlot(self, event):    # wxGlade: MyFrame.<event_handler>
        if    'TBS' in self.model:
            self.instr.open()
            self.instr.ask("*ESR?")
            self.instr.write("*CLS")
            self.instr.write("ACQUIRE:STATE STOP")
            self.instr.write("ACQUIRE:STOPAFTER SEQUENCE")
            self.instr.write("ACQUIRE:STATE RUN")
            self.instr.write("DATa:SOUrce CH1")
            print("Waiting for trigger...")
            print(self.instr.ask("BUSY?"))
            while self.instr.ask("BUSY?")=="1":
                pass
            self.instr.ask("*OPC?")
            print("Trigger received")

#Channel 1 ------------------------------------------------------------------

            self.instr.write("DATa:ENCdg ASCII")
            self.instr.write("DATa:WIDth 1")
            self.instr.write("DATa:STARt 1")
            self.instr.write("DATa:STOP 15")
            b=self.instr.ask("CURVe?")
            tscale = float(self.instr.ask('WFMPre:XINcr?'))
            tstart = float(self.instr.ask('WFMPre:XZEro?'))
            vscale1 = float(self.instr.ask('WFMPre:YMUlt?')) # volts / level
            voff1 = float(self.instr.ask('WFMPre:YZEro?')) # reference voltage
            vpos1 = float(self.instr.ask('WFMPre:YOFf?')) # reference position
(level)

            total_time = tscale * 15
```

```
                tstop = tstart + total_time
                x = np.linspace(tstart, tstop, num=15, endpoint=False)

                # vertical (voltage)
                bf1 = [float(i1) for i1 in b.split(',')]
                unscaled_wave1 = np.array(bf1, dtype='double') # data type
conversion
                y1 = (unscaled_wave1 - vpos1) * vscale1 + voff1

#Channel 2 ------------------------------------------------------------

                self.instr.write("DATa:SOUrce CH1")
                self.instr.write("DATa:ENCdg ASCII")
                self.instr.write("DATa:WIDth 1")
                self.instr.write("DATa:STARt 1")
                self.instr.write("DATa:STOP 15")
                b=self.instr.ask("CURVe?")
                vscale2 = float(self.instr.ask('WFMPre:YMUlt?')) # volts / level
                voff2 = float(self.instr.ask('WFMPre:YZEro?')) # reference voltage
                vpos2 = float(self.instr.ask('WFMPre:YOFf?')) # reference position
(level)

                # vertical (voltage)
                bf2 = [float(i2) for i2 in b.split(',')]
                unscaled_wave2 = np.array(bf2, dtype='double') # data type
conversion
                y2 = (unscaled_wave2 - vpos2) * vscale2 + voff2

#-------------------------------------------------------------------------------------------

            elif 'DPO' in self.model:
                self.instr.open()
                self.instr.ask("*ESR?")
                self.instr.write("*CLS")
```

```
                self.instr.write("ACQUIRE:STATE STOP")
                self.instr.write("ACQUIRE:STOPAFTER SEQUENCE")
                self.instr.write("ACQUIRE:STATE RUN")
                self.instr.write("DATa:SOUrce CH1")
                print("Waiting for trigger...")
                print(self.instr.ask("BUSY?"))
                while self.instr.ask("BUSY?")=="1":
                    pass
                self.instr.ask("*OPC?")
                print("Trigger received")

#Channel 1 ----------------------------------
                self.instr.write("DATa:ENCdg ASCII")
                self.instr.write("DATa:WIDth 1")
                self.instr.write("DATa:STARt 1")
                self.instr.write("DATa:STOP 20000")
                self.instr.write("WFMOutpre:BYT_Nr 16")
                b=self.instr.ask("CURVe?")
                a=self.instr.ask('WFMOutpre?')
                cap = a.split(";")
                y1 = [int(i1)*float(cap[14]) for i1 in b.split(",")]
                x = np.array(range(len(y1)))*float(cap[10])

#Channel 2 ----------------------------------------------
                self.instr.write("DATa:SOUrce CH2")
                self.instr.write("DATa:ENCdg ASCII")
                self.instr.write("DATa:WIDth 1")
                self.instr.write("DATa:STARt 1")
                self.instr.write("DATa:STOP 20000")
                self.instr.write("WFMOutpre:BYT_Nr 16")
                b=self.instr.ask("CURVe?")
                a=self.instr.ask('WFMOutpre?')
                cap = a.split(";")
                y2 = [int(i2)*float(cap[14]) for i2 in b.split(",")]
```

```
#Plot------------------------------------------------------------

        self.data_set.append(x, y1, y2)
        self.data_set.display_data(self)
        return
```

Now it is the turn of the Edit collapsible event handlers. The first one is another pillar of the program, “EAcquireSinglePlot”. This function is the one that captures de data from the oscilloscope and displays it on screen. It begins by discriminating though the model used as part of the configuration for the data capture and its display is different depending of the model. Either way both begin exactly the same by preparing the oscilloscope for the trigger and receiving it. Some extra commands are used for event cleaning and operation complete checks, like “*CLS” or “*OPC?” to make life easier for the oscilloscope. The first big differences come on the configuration of the data to extract. The biggest one in the experimental phase I carried out is that a TBS1k series can only reliably get 15 points to make the curve observed on the oscilloscope. This is far from being good and curves exported to the computer from this kind of oscilloscopes resemble nothing the curve seen on the oscilloscope’s screen, but this is not a fault that can be repaired through skilful software tempering, it is a hardware barrier. As a comparison, DPO3k series oscilloscopes can export curves with more than 20.000 points, curves that are a correct representation of the information obtained by the oscilloscope. Another weakness of TBS1k series oscilloscopes that makes the program longer and more complicated is the lack of a working command that exports the reference data (number of points and scale of the horizontal and vertical scale,...) to the computer, which forced me to add code lines to extract these reference data and do operations with it to get the actual data needed to plot a graph. Meanwhile this is easily solved in the DPO3k series by using the command “WFMOutpre?”. Both do need a split to compute the numbers obtained through “CURVe?”, which are unsurprisingly the values of the curve, or said in another way, the different values of y for each time x, which are channel specific unlike x. After getting a time “x” and curves “y1” from the channel 1 and “y2” from the channel 2, they are sent to be saved into the data set and call the “display_data” function to show a plot for channel 1 and another one for channel 2 on the UI.

```
def EDeletePlot(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.remove()
        self.data_set.display_data(self)
        return

def VFirst(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.display_data(self.data_set(0))
        return

def VLast(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.display_data(self.data_set(len(self.data_set.plots) - 1))
        return

def VBack(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.backward()
        self.data_set.display_data(self)
        return

def VForward(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.forward()
        self.data_set.display_data(self)
        return

def VJUmpTo(self, event):    # wxGlade: MyFrame.<event_handler>
        self.data_set.goto(self.plot_index)
        self.data_set.display_data(self)
        return
```

Then the “EDeletePlot” function comes. Along with all the View collapsible functions, they simply include calls to the “DataSet” class where all the necessary code for these options resides.

```
def TShownChannels(self, event):    # wxGlade: MyFrame.<event_handler>
```

```
        print("Input '1' to show channel 1 and 2 or input '0' not to.")
        dispch1=input("Show Channel 1?")
        self.instr.write("SELect:CH1 %s" %(dispch1))
        dispch2=input("Show Channel 2?")
        self.instr.write("SELect:CH1 %s" %(dispch2))
        return

    def TSelectChannelsCoupling(self, event):  # wxGlade: MyFrame.<event_handler>
        print("Input 'DC' or 'AC' (simple brackets included) for channel coupling and a + +  number in '1.0E0' format for channel probe.")
        ch1coupl=input("Channel 1 coupling?")
        print(ch1coupl)
        self.instr.write("CH1:COUPLing %s" %(ch1coupl))
        ch1probe=input("Channel 1 probe?")
        self.instr.write("CH1:PRObe %s" %(ch1probe))
        ch2coupl=input("Channel 2 coupling?")
        self.instr.write("CH2:COUPLing %s" %(ch2coupl))
        ch2probe=input("Channel 2 probe?")
        self.instr.write("CH2:PRObe %s" %(ch2probe))
        return

    def TVerticalScale(self, event):  # wxGlade: MyFrame.<event_handler>
        print("Input a vertical scale for Channel 1 and 2 in a '1.0E0' format.")
        ych1=input("Channel 1 vertical scale?")
        self.instr.write("CH1:VOLts %s" %(ych1))
        ych2=input("Channel 2 vertical scale?")
        self.instr.write("CH2:VOLts %s" %(ych2))
        return

    def THorizontalScale(self, event):  # wxGlade: MyFrame.<event_handler>
        print("Input the horizontal main axis scale in a '1.0E0' format.")
        xaxistime=input("Horizontal main axis scale?")
        self.instr.write("HORizontal:MAIn:SCAle %s" %(xaxistime))
```

```
        return

    def TTriggerType(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Select a trigger type like 'EDGE' (simple brackets included) and its + 
    + settings.")
        trigg=input("Trigger type?")
        self.instr.write("TRIGger:MAIn:TYPe %s" %(trigg))
        triggch=input("Trigger channel is 'CH1' or 'CH2'?")
        self.instr.write("TRIGger:MAIn:EDGE:SOUrce %s" %(triggch))
        trigglev=input("Trigger level in a '1.0E0' format is?")
        self.instr.write("TRIGger:MAIn:LEVel %s" %(trigglev))
        triggslope=input("Trigger slope is 'RISE' or 'FALL'?")
        self.instr.write("TRIGger:MAIn:EDGE:SLOpe %s" %(triggslope))
        triggcoupl=input("Trigger coupling is 'DC' or 'AC'?")
        self.instr.write("TRIGger:MAIn:EDGE:COUPLing %s" %(triggcoupl))
        return

    def TTriggerState(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'TTriggerState' not implemented!")
        triggstat=input("Osciloscope state is Run/Stop 'RUNSTOP' or Trigger + 
    +   'SEQUENCE'?")
        self.instr.write("ACQuire:STOPAfter %s" %(triggstat))
        startstat=input("Trigger state is ready '1' or not ready '0'?")
        self.instr.write("ACQuire:STATE %s" %(startstat))
        return
```

The Tools collapsible event handlers also have much of the work done as they are the changes tested in Annex's 1: save_launch_load.py second section. In the Tools collapsible they have simply been arranged into reasonable groups and when clicking one the terminal acts as the receiver of the user's inputs and as a guide of how to write the inputs so that the SCPI language of the oscilloscope will understand them. To finish the "ElementalFrame" class we have the event handler for "HAbout", the only button in the Help collapsible. It has no significant information and was only added to

finish the usual File/Edit/View/Tools/Help style menu bar because the terminal handholds the user through the use of the program.

```
class ElementalApp(wx.App):
    def OnInit(self):
        self.frame = ElementalFrame(None, wx.ID_ANY, "")
        self.SetTopWindow(self.frame)
        self.frame.Show()
        return True

if __name__ == "__main__":
    app = ElementalApp(0)
    app.MainLoop()
```

Finally the last class of the Annex 3: tfg_program.py is reached, “ElementalApp”. This class is automatically created by WxGlade and as its good programming practice to have this classes created by WxGlade in the program to be run, I copied it but did not change a single thing about it.

2.3.4 Practical test of the finished program in the laser experiment

With the program functioning on the TBS1052 oscilloscope I was allowed into the Optics laboratory to see if the program worked on actual experiment conditions overseen by my tutor and the laboratory's responsible. We used both my personal laptop and the laboratory's Raspberry Pi (which needed to have the necessary libraries downloaded into it) to run the program. First we set up the experiment as explained in 'Approach and decision about alternative solution' by setting up the piezoelectric in a sample holder so that it will be hit be the laser when shot. The oscilloscope, a DPO3054, was connected to the laser to catch the warning signal it gives shortly before shooting the laser through channel 1 (this one would be used as the trigger for the oscilloscope) and to the piezoelectric through channel 2. Then, for the Raspberry Pi we also set up the peripherals needed to use it (screen, keyboard and mouse) and give power to the laser and the computers used. The Raspberry Pi was the one that had both the TTL signal controller and the software needed to give the shooting order to the laser so it was used in the two test that we did.

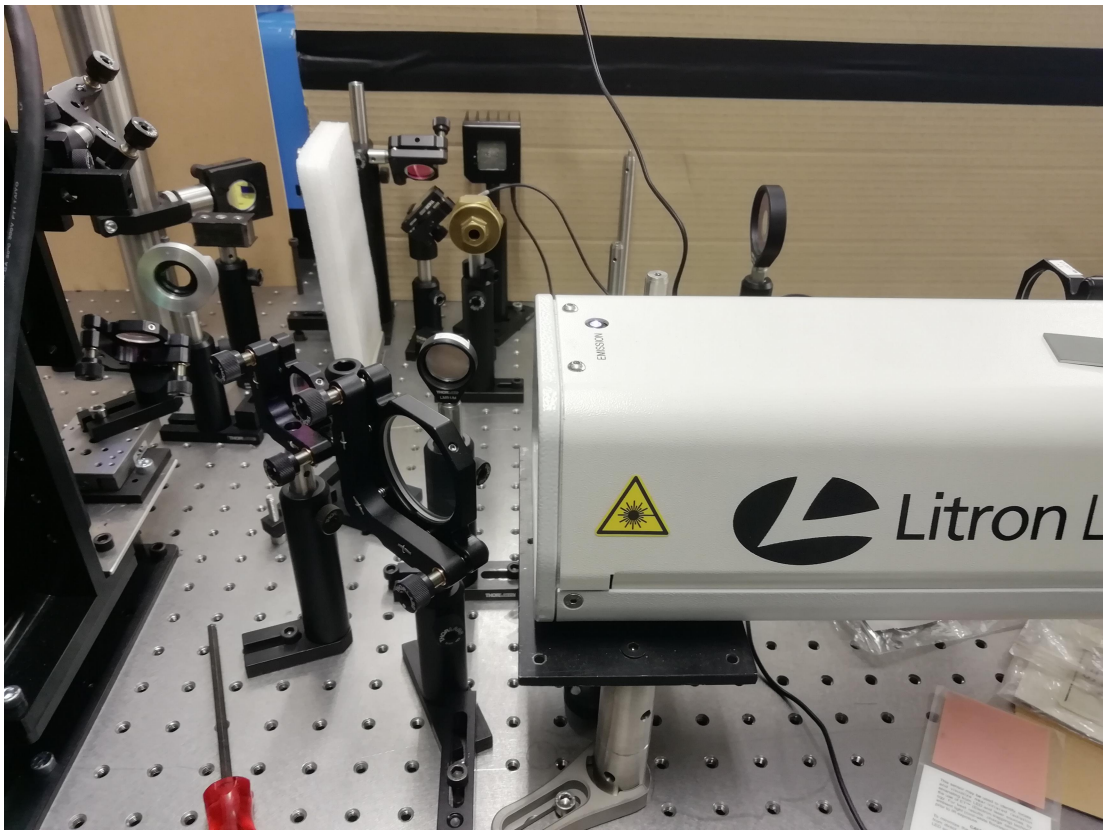

Figure 11: Experiment close up, laser and sample holder.

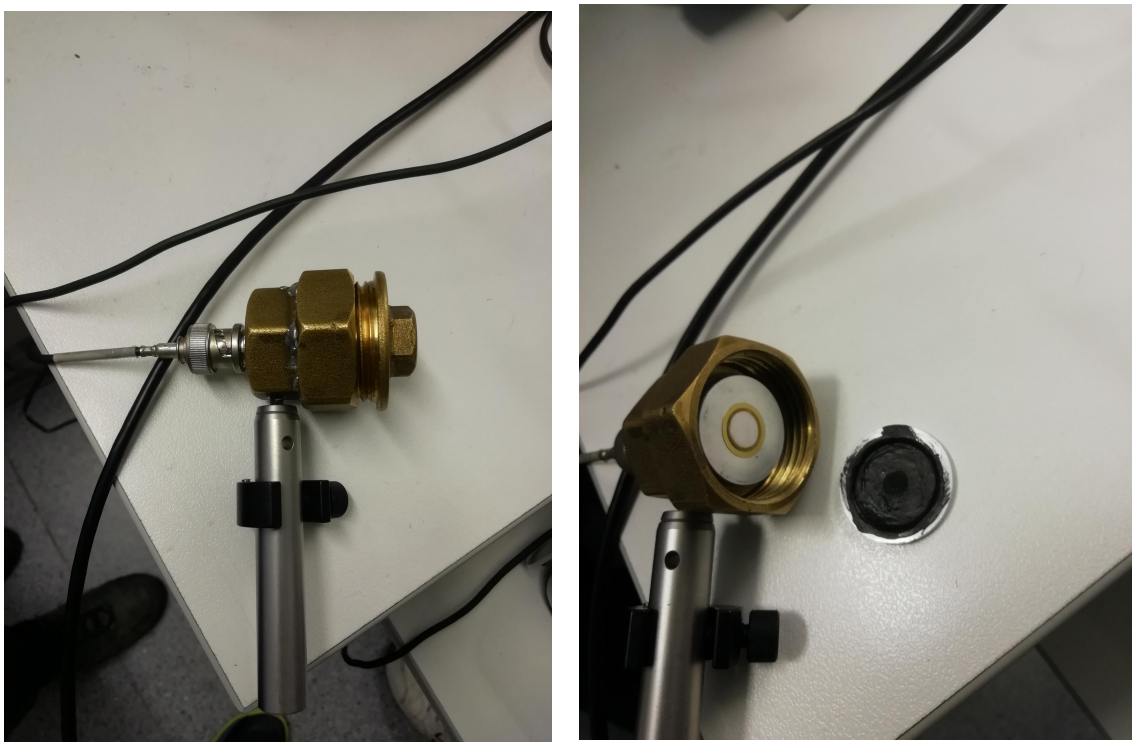

Figure 12 and 13: Sample holder closed and open (with visible piezoelectric in F.13).

The first test was done with my laptop running the program. The oscilloscope was connected with a type B USB to the computer and the settings were changed through the Tools collapsible options. Then using the Edit collapsible option Acquire Single Plot a set of data was successfully acquired, navigated through the View collapsible options, saved and loaded correctly.

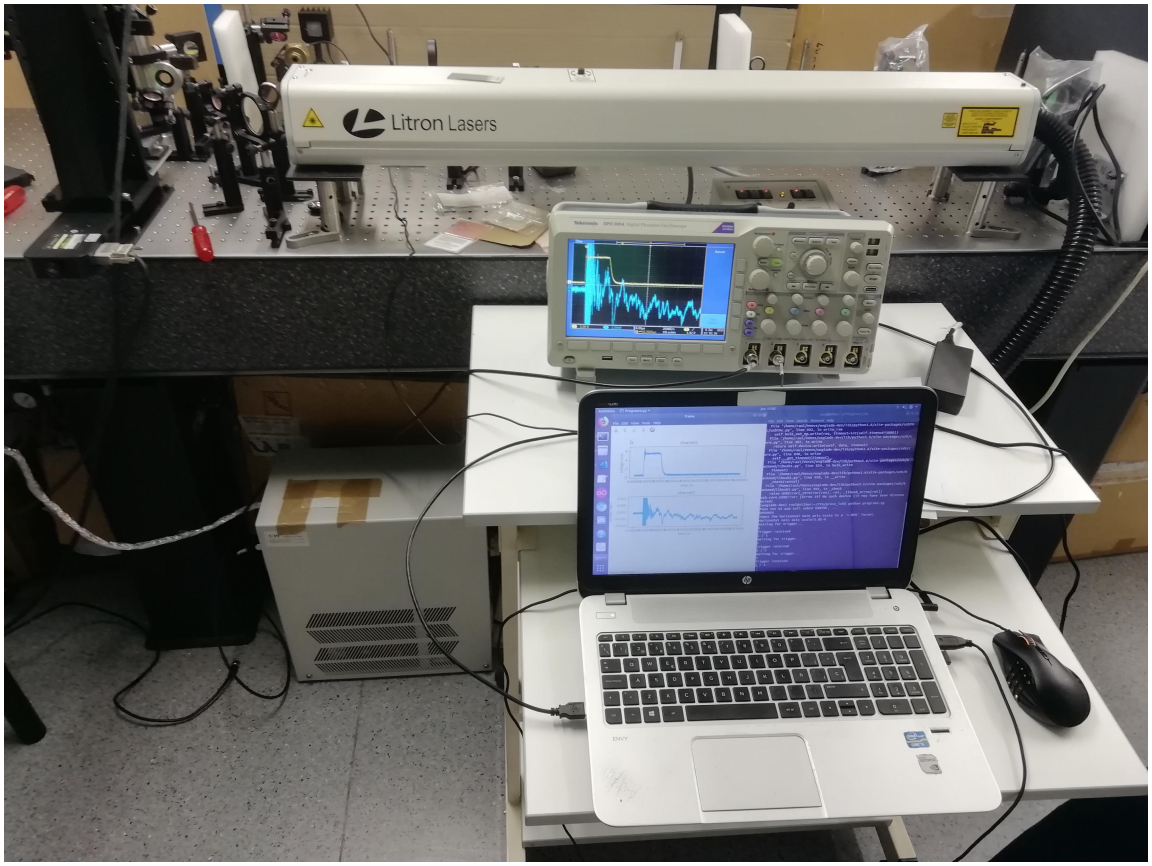

Figure 14: Experiment setup.

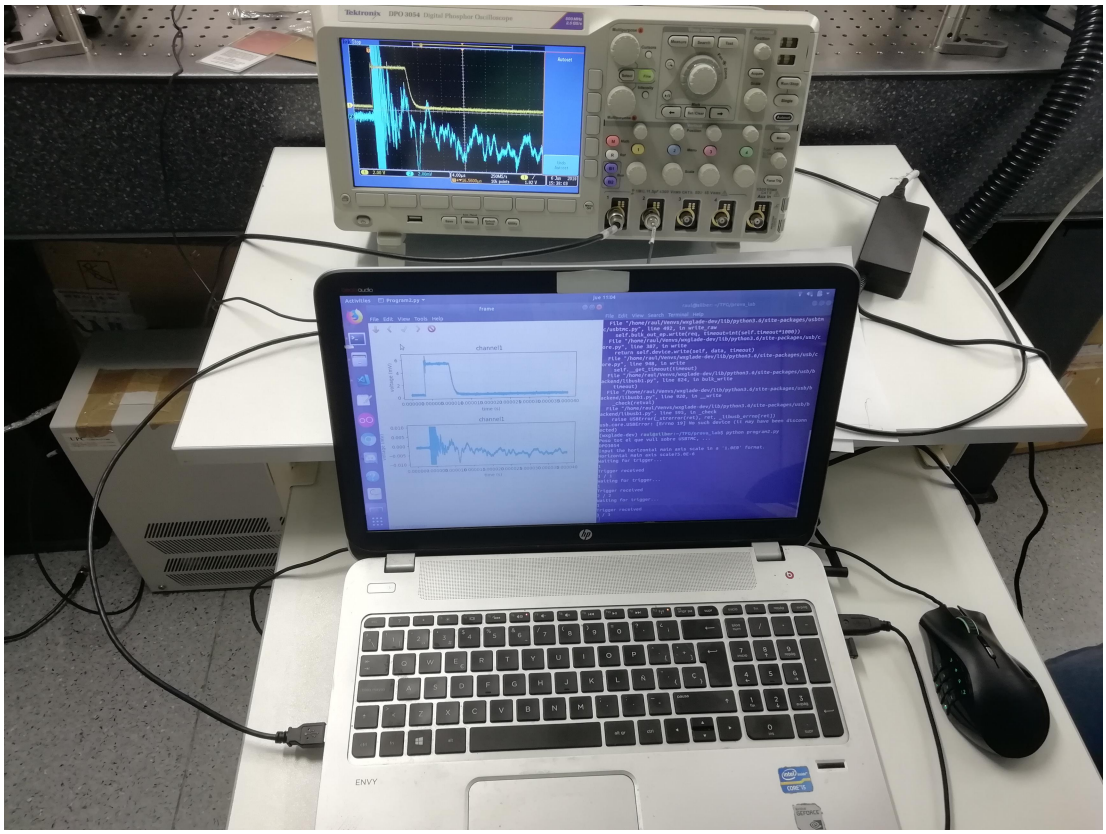

Figure 15: PC and oscilloscope, connected with a type B USB (to the left)

The second test had the Raspberry Pi running the program. Just as the first test, settings were changed settings through Tools and obtained a data set with Acquire Single Plot. The rest of options were proven to work as intended and the test was carried out successfully as well.

Figure 16: Raspberry setup (with TTL [to the right], USB [center] and VGA [to the left] connections).

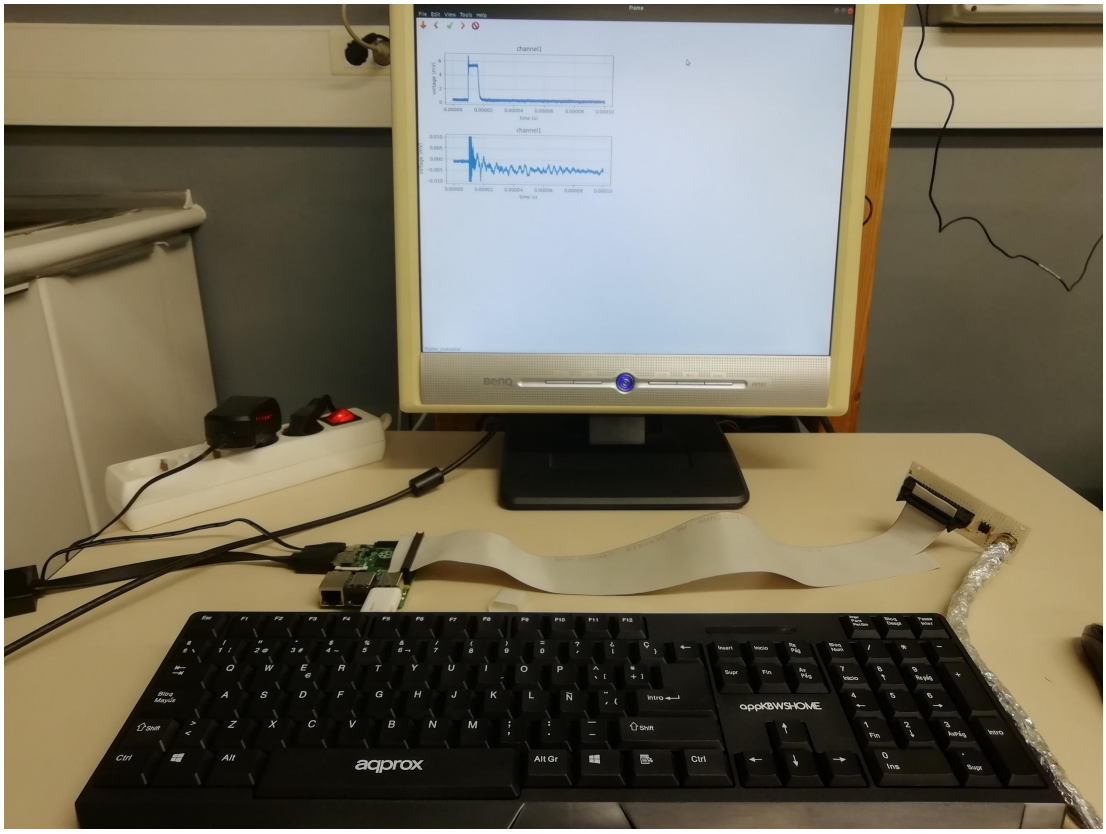

Figure 17: raspberry Pi showing extracted data

3. Results summary

3.1 Conclusions

At the end of this project I think it can be said that the objectives of the project have been successfully achieved. A program with graphical interface has been created and it has the functionalities for saving and loading data, for remotely operating the oscilloscope connected to the computer running the program and for restoring the settings prior to the experiment's setup, all in a single program.

The budget has also been kept to a minimum by using open source software and even the simulation circuit used to run the first test of the program is quite the cheap piece of equipment.

However that does not mean that cheap is always better. As I have experienced during this project, even if the program is cheap and simple, it cannot solve the limitations that hardware present. This is said due to the inability to show a proper plot of the captured signal in the quite old TBS1052 oscilloscope. If the equipment has low performance, the program will not improve it.

3.2 Continuation project suggestions

At the end of the project it is quite easy to find things to improve as this program is still a hatchling of what it could become.

For the graphical interface, the UI still has much space to be used. A Tool Bar and a Status Bar could be a good first step. Also, right now it is a program designed to be used in tandem with the Terminal, redirecting all of its uses unto the WxGlade window would make for a cleaner and more organised view.

For the code, more settings to be altered could be added and perhaps the most interesting addition would be to create a way for the program to automatically detect the oscilloscope used and write down its ID where it needs to be inside the program.

For the data, more files types available for saving would be an improvement. This could be tied into adding options or creating a second program to do some light numerical data treatment.

4. Bibliography

[1] Vishay. 4N25, 4N26, 4N27, 4N28 Optocoupler, Phototransistor Output, with Base Connection [pdf datasheet]. Available: https://www.vishay.com/docs/83725/4n25.pdf Last checked:10/6/19

[2] Texas Instruments. LMx24-N, LM2902-N Low-Power, Quad-Operational Amplifiers [pdf datasheet]. Available: http://www.ti.com/lit/ds/symlink/lm324-n.pdf Last checked:10/6/19

[3] Texas Instruments. LM555 Timer [pdf datasheet]. Available: http://www.ti.com/lit/ds/symlink/lm555.pdf Last checked:10/6/19

[4] Tektronix. (23 Sep 2015). TBS1000B and TBS1000B-EDU Series Manual Oscilloscopes User Manual [pdf Manual]. Available: https://www.tek.com/manual/downloads?product_series=TBS1000&manual_type=301 Last checked:10/6/19

[5] Tektronix. (03 Jul 2014). TBS1000B, TBS1000, TDS200, TDS1000/2000, TDS1000B/2000B, TDS1000C-EDU/TDS2000C, TDS2024C, TPS2000/TPS2000B Series Digital Oscilloscope Programmer Manual [pdf Manual]. Available: https://www.tek.com/manual/downloads?product_series=TBS1000&manual_type=306 Last checked:10/6/19

[6] Tektronix. (year, month day). MSO3000 and DPO3000 Series Digital Phosphor Oscilloscopes User Manual [pdf Manual]. Available: https://www.tek.com/manual/downloads?product_series=DPO3000&manual_type=301 Last checked:10/6/19

[7] Tektronix. (22 Mar 2012). MSO3000 and DPO3000 Series Digital Phosphor Oscilloscopes Programmer Manual [pdf Manual]. Available: https://www.tek.com/manual/downloads?product_series=DPO3000&manual_type=306 Last checked:10/6/19

5. Annexes

5.1 Annex 1: save_launch_load.py

```
import usbtmc
import time

instr = usbtmc.Instrument(0x699, 0x368)
scopename = instr.ask("*IDN?")
scopename = scopename.split(',')
model = scopename[1]
instr.open()
prevtig=False

#Saving altered variables

trigg=instr.ask("TRIGger:MAIn:TYPe?")
triggch=instr.ask("TRIGger:MAIn:EDGE:SOUrce?")
trigglev=instr.ask("TRIGger:MAIn:LEVel?")
if trigg=="EDGE":
	prevtig=True
	triggslope=instr.ask("TRIGger:MAIn:EDGE:SLOpe?")
	triggcoupl=instr.ask("TRIGger:MAIn:EDGE:COUPLing?")

time.sleep(0.5)

dispch1=instr.ask("SELect:CH1?")
dispch2=instr.ask("SELect:CH2?")

time.sleep(0.5)

ch1coupl=instr.ask("CH1:COUPLing?")
ch1probe=instr.ask("CH1:PRObe?")
ch2coupl=instr.ask("CH2:COUPLing?")
ch2probe=instr.ask("CH2:PRObe?")

time.sleep(0.5)

ych1=instr.ask("CH1:VOLts?")
ych2=instr.ask("CH2:VOLts?")

time.sleep(0.5)

xaxistime=instr.ask("HORizontal:MAIn:SCAle?")

time.sleep(0.5)

triggstat=instr.ask("ACQuire:STOPAfter?")
startstat=instr.ask("ACQuire:STATE?")
```

```
time.sleep(0.5)

#Begin setup

if   'TBS' in model:
        instr.open()
        instr.ask("*ESR?")
        instr.write("*CLS")
        instr.write("ACQUIRE:STATE STOP")
        instr.write("ACQUIRE:STOPAFTER SEQUENCE")
        instr.write("ACQUIRE:STATE RUN")
        instr.write("DATa:SOUrce CH1")
        print("Waiting for trigger...")
        print(instr.ask("BUSY?"))
        while instr.ask("BUSY?")=="1":
                pass
        instr.ask("*OPC?")
        print("Trigger received")

elif 'DPO' in model:
        instr.open()
        instr.ask("*ESR?")
        instr.write("*CLS")
        instr.write("ACQUIRE:STATE STOP")
        instr.write("ACQUIRE:STOPAFTER SEQUENCE")
        instr.write("ACQUIRE:STATE RUN")
        instr.write("DATa:SOUrce CH1")
        print("Waiting for trigger...")
        print(instr.ask("BUSY?"))
        while instr.ask("BUSY?")=="1":
                pass
        instr.ask("*OPC?")
        print("Trigger received")

#Load initial saved settings

instr.write("TRIGger:MAIn:TYPe %s" %(trigg))
instr.write("TRIGger:MAIn:EDGE:SOUrce %s" %(triggch))
instr.write("TRIGger:MAIn:LEVel %s" %(trigglev))
if prevtig==True:
        instr.write("TRIGger:MAIn:EDGE:SLOpe %s" %(triggslope))
        instr.write("TRIGger:MAIn:EDGE:COUPLing %s" %(triggcoupl))

time.sleep(0.5)

instr.write("SELect:CH1 %s" %(dispch1))
instr.write("SELect:CH2 %s" %(dispch2))

time.sleep(0.5)
```

```
instr.write("CH1:COUPLing %s" %(ch1coupl))
instr.write("CH1:PRObe %s" %(ch1probe))
instr.write("CH2:COUPLing %s" %(ch2coupl))
instr.write("CH2:PRObe %s" %(ch2probe))

time.sleep(0.5)

instr.write("CH1:VOLts %s" %(ych1))
instr.write("CH2:VOLts %s" %(ych2))

time.sleep(0.5)

instr.write("HORizontal:MAIn:SCAle %s" %(xaxistime))

time.sleep(0.5)

instr.write("ACQuire:STOPAfter %s" %(triggstat))
instr.write("ACQuire:STATE %s" %(startstat))
```

5.2 Annex 2: wxglade_frame_gui.py

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
#
# generated by wxGlade 0.9.3 on Sun Jun  9 22:05:51 2019
#

import wx

# begin wxGlade: dependencies
# end wxGlade

# begin wxGlade: extracode
# end wxGlade


class MyFrame(wx.Frame):
    def __init__(self, *args, **kwds):
        # begin wxGlade: MyFrame.__init__
        kwds["style"] = kwds.get("style", 0) | wx.DEFAULT_FRAME_STYLE
        wx.Frame.__init__(self, *args, **kwds)
        self.SetSize((400, 336))

        # Menu Bar
        self.frame_menubar = wx.MenuBar()
        wxglade_tmp_menu = wx.Menu()
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "New", "")
        self.Bind(wx.EVT_MENU, self.FNew, id=item.GetId())
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Open", "")
        self.Bind(wx.EVT_MENU, self.FOpen, id=item.GetId())
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Save", "")
        self.Bind(wx.EVT_MENU, self.FSave, id=item.GetId())
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Save as...", "")
        self.Bind(wx.EVT_MENU, self.FSaveAs, id=item.GetId())
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Restore Initial Settings",
"")
        self.Bind(wx.EVT_MENU, self.FRestore, id=item.GetId())
        self.frame_menubar.Append(wxglade_tmp_menu, "File")
        wxglade_tmp_menu = wx.Menu()
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Acquire Single Plot",
"")
        self.Bind(wx.EVT_MENU, self.EAcquireSinglePlot, id=item.GetId())
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Delete Plot", "")
        self.Bind(wx.EVT_MENU, self.EDeletePlot, id=item.GetId())
        self.frame_menubar.Append(wxglade_tmp_menu, "Edit")
        wxglade_tmp_menu = wx.Menu()
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "First", "")
        self.Bind(wx.EVT_MENU, self.VFirst, id=item.GetId())
        item = wxglade_tmp_menu.Append(wx.ID_ANY, "Last", "")
```

```
            self.Bind(wx.EVT_MENU, self.VLast, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Back", "")
            self.Bind(wx.EVT_MENU, self.VBack, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Forward", "")
            self.Bind(wx.EVT_MENU, self.VForward, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Jump to", "")
            self.Bind(wx.EVT_MENU, self.VJUmpTo, id=item.GetId())
            self.frame_menubar.Append(wxglade_tmp_menu, "View")
            wxglade_tmp_menu = wx.Menu()
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Shown Channels", "")
            self.Bind(wx.EVT_MENU, self.TShownChannels, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Select Channels' Couplings", "")
            self.Bind(wx.EVT_MENU, self.TSelectChannelsCoupling, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Vertical Scale", "")
            self.Bind(wx.EVT_MENU, self.TVerticalScale, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Horizontal Scale", "")
            self.Bind(wx.EVT_MENU, self.THorizontalScale, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Trigger Type", "")
            self.Bind(wx.EVT_MENU, self.TTriggerType, id=item.GetId())
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "Trigger State", "")
            self.Bind(wx.EVT_MENU, self.TTriggerState, id=item.GetId())
            self.frame_menubar.Append(wxglade_tmp_menu, "Tools")
            wxglade_tmp_menu = wx.Menu()
            item = wxglade_tmp_menu.Append(wx.ID_ANY, "About", "")
            self.Bind(wx.EVT_MENU, self.HAbout, id=item.GetId())
            self.frame_menubar.Append(wxglade_tmp_menu, "Help")
            self.SetMenuBar(self.frame_menubar)
            # Menu Bar end
            self.panel_1 = wx.Panel(self, wx.ID_ANY)

            self.__set_properties()
            self.__do_layout()

            # end wxGlade

        def __set_properties(self):
            # begin wxGlade: MyFrame.__set_properties
            self.SetTitle("frame")
            # end wxGlade

        def __do_layout(self):
            # begin wxGlade: MyFrame.__do_layout
            sizer_1 = wx.BoxSizer(wx.VERTICAL)
            sizer_1.Add(self.panel_1, 1, wx.EXPAND, 0)
            self.SetSizer(sizer_1)
            self.Layout()
            # end wxGlade
```

```
def MFile(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'MFile' not implemented!")
        event.Skip()

def FNew(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'FNew' not implemented!")
        event.Skip()

def FOpen(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'FOpen' not implemented!")
        event.Skip()

def FSave(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'FSave' not implemented!")
        event.Skip()

def FSaveAs(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'FSaveAs' not implemented!")
        event.Skip()

def FRestore(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'FRestore' not implemented!")
        event.Skip()

def MEdit(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'MEdit' not implemented!")
        event.Skip()

def EAcquireSinglePlot(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'EAcquireSinglePlot' not implemented!")
        event.Skip()

def EDeletePlot(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'EDeletePlot' not implemented!")
        event.Skip()

def MView(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'MView' not implemented!")
        event.Skip()

def VFirst(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'VFirst' not implemented!")
        event.Skip()

def VLast(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'VLast' not implemented!")
        event.Skip()

def VBack(self, event):    # wxGlade: MyFrame.<event_handler>
        print("Event handler 'VBack' not implemented!")
```

```
            event.Skip()

        def VForward(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'VForward' not implemented!")
            event.Skip()

        def VJUmpTo(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'VJUmpTo' not implemented!")
            event.Skip()

        def MTools(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'MTools' not implemented!")
            event.Skip()

        def TShownChannels(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'TShownChannels' not implemented!")
            event.Skip()

        def TSelectChannelsCoupling(self, event):    # wxGlade:
MyFrame.<event_handler>
            print("Event handler 'TSelectChannelsCoupling' not implemented!")
            event.Skip()

        def TVerticalScale(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'TVerticalScale' not implemented!")
            event.Skip()

        def THorizontalScale(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'THorizontalScale' not implemented!")
            event.Skip()

        def TTriggerType(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'TTriggerType' not implemented!")
            event.Skip()

        def TTriggerState(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'TTriggerState' not implemented!")
            event.Skip()

        def HAbout(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'HAbout' not implemented!")
            event.Skip()

# end of class MyFrame

class MyApp(wx.App):
        def OnInit(self):
            self.frame = MyFrame(None, wx.ID_ANY, "")
            self.SetTopWindow(self.frame)
            self.frame.Show()
```

```
            return True

# end of class MyApp

if __name__ == "__main__":
    app = MyApp(0)
    app.MainLoop()
```

5.3 Annex 3: tfg_program.py

```
import wx
from struct import unpack
import usbtmc
import time
import numpy as np
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.figure import Figure
from wxglade_frame_gui import MyFrame
import pickle as pk
import os

class DataPlot:
    def __init__(self, t, y1, y2,):
        self.t = t
        self.y1 = y1
        self.y2 = y2
        return

class DataSet:
    def __init__(self):
        self.plots = []
        self.display = -1
        return
    def append(self, t, y1, y2):
        plot = DataPlot(t, y1, y2)
        self.plots.append(plot)
        self.display = len(self.plots) - 1
        self.print_position()
        return
    def remove(self):
        if self.display < 0:
            return
        self.plots = self.plots[:self.display]+self.plots[self.display+1:]
        if self.display == len(self.plots):
            self.display = self.display - 1
        self.print_position()
        return
    def forward(self):
        if self.display < 0:
            return
        if self.display < len(self.plots) - 1:
            self.display = self.display + 1
        self.print_position()
        return
    def backward(self):
        if self.display > 0:
            self.display = self.display - 1
```

```
            self.print_position()
            return
        def goto(self, plot_index):
            if self.display < 0:
                return
            plot_index = int(input("Introduce the postion of the plot:"))
            if plot_index > 0 and plot_index < len(self.plots) + 1:
                self.display = plot_index - 1
            self.print_position()
            return
        def print_position(self):
            print(self.display + 1, "/", len(self.plots))
            return
        def display_data(self, frame):
            if self.display < 0:
                return
            X = self.plots[self.display].t
            Y = self.plots[self.display].y1
            Z = self.plots[self.display].y2
            frame.ax1.cla()
            frame.ax2.cla()
            frame.ax1.plot(X, Y)
            frame.ax1.set(xlabel='time (s)', ylabel='voltage (V)', title='channel1')
            frame.ax1.grid()
            frame.ax2.plot(X, Z)
            frame.ax2.set(xlabel='time (s)', ylabel='voltage (V)', title='channel2')
            frame.ax2.grid()
            frame.canvas.draw()
            return

class ElementalFrame(MyFrame):
        def __init__(self, *args, **kwds):
            MyFrame.__init__(self, *args, **kwds)
            print("Welcome.")
            self.figure = Figure()
            self.ax1 = self.figure.add_subplot(211)
            self.ax2 = self.figure.add_subplot(212,sharex=self.ax1)
            self.figure.subplots_adjust(hspace=0.6, wspace=0.4)
            self.canvas = FigureCanvas(self.panel_1, -1, self.figure)
            self.data_set = DataSet()
            self.data_path = ""

            self.instr = usbtmc.Instrument(0x699, 0x415) #DPO 3054
            #self.instr = usbtmc.Instrument(0x699, 0x368) #TBS 1052B
            self.scopename = self.instr.ask("*IDN?")
            self.scopename = self.scopename.split(',')
            self.model = self.scopename[1]
            self.instr.timeout = 15
            self.instr.open()
            self.trigg=self.instr.ask("TRIGger:MAIn:TYPe?")
```

```
            self.triggch=self.instr.ask("TRIGger:MAIn:EDGE:SOUrce?")
            self.trigglev=self.instr.ask("TRIGger:MAIn:LEVel?")
            if self.trigg=="EDGE":
                self.prevtig=True
                self.triggslope=self.instr.ask("TRIGger:MAIn:EDGE:SLOpe?")
                self.triggcoupl=self.instr.ask("TRIGger:MAIn:EDGE:COUPLing?")

            time.sleep(0.5)

            self.dispch1=self.instr.ask("SELect:CH1?")
            self.dispch2=self.instr.ask("SELect:CH2?")

            time.sleep(0.5)

            self.ch1coupl=self.instr.ask("CH1:COUPLing?")
            self.ch1probe=self.instr.ask("CH1:PRObe?")
            self.ch2coupl=self.instr.ask("CH2:COUPLing?")
            self.ch2probe=self.instr.ask("CH2:PRObe?")

            time.sleep(0.5)

            self.ych1=self.instr.ask("CH1:VOLts?")
            self.ych2=self.instr.ask("CH2:VOLts?")

            time.sleep(0.5)

            self.xaxistime=self.instr.ask("HORizontal:MAIn:SCAle?")

            time.sleep(0.5)

            self.triggstat=self.instr.ask("ACQuire:STOPAfter?")
            self.startstat=self.instr.ask("ACQuire:STATE?")

            time.sleep(0.5)
            return

        def FNew(self, event):    # wxGlade: MyFrame.<event_handler>
            self.data_set.__init__()
            self.data_path = ""
            self.ax1.cla()
            self.ax2.cla()
            self.canvas.draw()
            return

        def FOpen(self, event):    # wxGlade: MyFrame.<event_handler>
            wildcard = "Pickle file (*.pkl)|*.pkl|" "All files (*.*)|*.*"
            dlg = wx.FileDialog(self, message="Choose a file", defaultDir=os.getcwd(),
defaultFile="", wildcard=wildcard, style=wx.FD_OPEN)
            if dlg.ShowModal() == wx.ID_OK:
                paths = dlg.GetPaths()
```

```
                print("You chose the following file(s):")
                for path in paths:
                    print(path)
                self.data_path = paths[0]
            dlg.Destroy()
            inputfile = open(self.data_path, 'rb')
            self.data_set = pk.load(inputfile)
            inputfile.close()
            self.data_set.display_data(self)
            return

        def FSave(self, event):    # wxGlade: MyFrame.<event_handler>
            if self.data_path == "":
                self.FSaveAs(event)
            else:
                outputfile = open(self.data_path, 'wb')
                pk.dump(self.data_set, outputfile)
                outputfile.close()
            return

        def FSaveAs(self, event):    # wxGlade: MyFrame.<event_handler>
            wildcard = "Pickle file (*.pkl)|*.pkl|" "All files (*.*)|*.*"
            dlg = wx.FileDialog(
            self, message="Save file as ...", defaultDir=os.getcwd(), wildcard=wildcard,
defaultFile="data.pkl", style=wx.FD_SAVE)
            if dlg.ShowModal() == wx.ID_OK:
                self.data_path = dlg.GetPath()
                print("You chose the following filename: %s" % self.data_path)
            dlg.Destroy()
            outputfile = open(self.data_path, 'wb')
            pk.dump(self.data_set, outputfile)
            outputfile.close()
            return

        def FRestore(self, event):    # wxGlade: MyFrame.<event_handler>
            self.instr.write("TRIGger:MAIn:TYPe %s" %(self.trigg))
            self.instr.write("TRIGger:MAIn:EDGE:SOUrce %s" %(self.triggch))
            self.instr.write("TRIGger:MAIn:LEVel %s" %(self.trigglev))
            if self.prevtig==True:
                self.instr.write("TRIGger:MAIn:EDGE:SLOpe %s" %(self.triggslope))

self.instr.write("TRIGger:MAIn:EDGE:COUPLing %s" %(self.triggcoupl))

            time.sleep(0.5)

            self.instr.write("SELect:CH1 %s" %(self.dispch1))
            self.instr.write("SELect:CH2 %s" %(self.dispch2))

            time.sleep(0.5)
```

```
            self.instr.write("CH1:COUPLing %s" %(self.ch1coupl))
            self.instr.write("CH1:PRObe %s" %(self.ch1probe))
            self.instr.write("CH2:COUPLing %s" %(self.ch2coupl))
            self.instr.write("CH2:PRObe %s" %(self.ch2probe))

            time.sleep(0.5)

            self.instr.write("CH1:VOLts %s" %(self.ych1))
            self.instr.write("CH2:VOLts %s" %(self.ych2))

            time.sleep(0.5)

            self.instr.write("HORizontal:MAIn:SCAle %s" %(self.xaxistime))

            time.sleep(0.5)

            self.instr.write("ACQuire:STOPAfter %s" %(self.triggstat))
            self.instr.write("ACQuire:STATE %s" %(self.startstat))
            return

        def EAcquireSinglePlot(self, event):    # wxGlade: MyFrame.<event_handler>
            if    'TBS' in self.model:
                  self.instr.open()
                  self.instr.ask("*ESR?")
                  self.instr.write("*CLS")
                  self.instr.write("ACQUIRE:STATE STOP")
                  self.instr.write("ACQUIRE:STOPAFTER SEQUENCE")
                  self.instr.write("ACQUIRE:STATE RUN")
                  self.instr.write("DATa:SOUrce CH1")
                  print("Waiting for trigger...")
                  print(self.instr.ask("BUSY?"))
                  while self.instr.ask("BUSY?")=="1":
                        pass
                  self.instr.ask("*OPC?")
                  print("Trigger received")

#Channel 1 ------------------------------------------------------------------

                  self.instr.write("DATa:ENCdg ASCII")
                  self.instr.write("DATa:WIDth 1")
                  self.instr.write("DATa:STARt 1")
                  self.instr.write("DATa:STOP 15")
                  b=self.instr.ask("CURVe?")
                  tscale = float(self.instr.ask('WFMPre:XINcr?'))
                  tstart = float(self.instr.ask('WFMPre:XZEro?'))
                  vscale1 = float(self.instr.ask('WFMPre:YMUlt?')) # volts / level
                  voff1 = float(self.instr.ask('WFMPre:YZEro?')) # reference voltage
                  vpos1  =  float(self.instr.ask('WFMPre:YOFf?'))  #  reference  position
(level)
```

```
                total_time = tscale * 15
                tstop = tstart + total_time
                x = np.linspace(tstart, tstop, num=15, endpoint=False)

                # vertical (voltage)
                bf1 = [float(i1) for i1 in b.split(',')]
                unscaled_wave1 = np.array(bf1, dtype='double') # data type
conversion
                y1 = (unscaled_wave1 - vpos1) * vscale1 + voff1

#Channel 2 ------------------------------------------------------------

                self.instr.write("DATa:SOUrce CH1")
                self.instr.write("DATa:ENCdg ASCII")
                self.instr.write("DATa:WIDth 1")
                self.instr.write("DATa:STARt 1")
                self.instr.write("DATa:STOP 15")
                b=self.instr.ask("CURVe?")
                vscale2 = float(self.instr.ask('WFMPre:YMUlt?')) # volts / level
                voff2 = float(self.instr.ask('WFMPre:YZEro?')) # reference voltage
                vpos2 = float(self.instr.ask('WFMPre:YOFf?')) # reference position
(level)

                # vertical (voltage)
                bf2 = [float(i2) for i2 in b.split(',')]
                unscaled_wave2 = np.array(bf2, dtype='double') # data type
conversion
                y2 = (unscaled_wave2 - vpos2) * vscale2 + voff2

#-------------------------------------------------------------------------------------------

            elif 'DPO' in self.model:
                self.instr.open()
                self.instr.ask("*ESR?")
                self.instr.write("*CLS")
                self.instr.write("ACQUIRE:STATE STOP")
                self.instr.write("ACQUIRE:STOPAFTER SEQUENCE")
                self.instr.write("ACQUIRE:STATE RUN")
                self.instr.write("DATa:SOUrce CH1")
                print("Waiting for trigger...")
                print(self.instr.ask("BUSY?"))
                while self.instr.ask("BUSY?")=="1":
                    pass
                self.instr.ask("*OPC?")
                print("Trigger received")

#Channel 1 ----------------------------------
                self.instr.write("DATa:ENCdg ASCII")
                self.instr.write("DATa:WIDth 1")
                self.instr.write("DATa:STARt 1")
```

```
                self.instr.write("DATa:STOP 20000")
                self.instr.write("WFMOutpre:BYT_Nr 16")
                b=self.instr.ask("CURVe?")
                a=self.instr.ask('WFMOutpre?')
                cap = a.split(";")
                y1 = [int(i1)*float(cap[14]) for i1 in b.split(",")]
                x = np.array(range(len(y1)))*float(cap[10])

#Channel 2 ------------------------------------------------
                self.instr.write("DATa:SOUrce CH2")
                self.instr.write("DATa:ENCdg ASCII")
                self.instr.write("DATa:WIDth 1")
                self.instr.write("DATa:STARt 1")
                self.instr.write("DATa:STOP 20000")
                self.instr.write("WFMOutpre:BYT_Nr 16")
                b=self.instr.ask("CURVe?")
                a=self.instr.ask('WFMOutpre?')
                cap = a.split(";")
                y2 = [int(i2)*float(cap[14]) for i2 in b.split(",")]

#Plot---------------------------------------------------------------

            self.data_set.append(x, y1, y2)
            self.data_set.display_data(self)
            return

        def EDeletePlot(self, event):    # wxGlade: MyFrame.<event_handler>
            self.data_set.remove()
            self.data_set.display_data(self)
            return

        def VFirst(self, event):    # wxGlade: MyFrame.<event_handler>
            self.data_set.display_data(self.data_set(0))
            return

        def VLast(self, event):    # wxGlade: MyFrame.<event_handler>
            self.data_set.display_data(self.data_set(len(self.data_set.plots) - 1))
            return

        def VBack(self, event):    # wxGlade: MyFrame.<event_handler>
            self.data_set.backward()
            self.data_set.display_data(self)
            return

        def VForward(self, event):    # wxGlade: MyFrame.<event_handler>
            self.data_set.forward()
            self.data_set.display_data(self)
            return

        def VJUmpTo(self, event):    # wxGlade: MyFrame.<event_handler>
```

```
            self.data_set.goto(self.plot_index)
            self.data_set.display_data(self)
            return

        def TShownChannels(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Input '1' to show channel 1 and 2 or input '0' not to.")
            dispch1=input("Show Channel 1?")
            self.instr.write("SELect:CH1 %s" %(dispch1))
            dispch2=input("Show Channel 2?")
            self.instr.write("SELect:CH1 %s" %(dispch2))
            return

        def TSelectChannelsCoupling(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Input 'DC' or 'AC' (simple brackets included) for channel coupling and a number in '1.0E0' format for channel probe.")
            ch1coupl=input("Channel 1 coupling?")
            print(ch1coupl)
            self.instr.write("CH1:COUPLing %s" %(ch1coupl))
            ch1probe=input("Channel 1 probe?")
            self.instr.write("CH1:PRObe %s" %(ch1probe))
            ch2coupl=input("Channel 2 coupling?")
            self.instr.write("CH2:COUPLing %s" %(ch2coupl))
            ch2probe=input("Channel 2 probe?")
            self.instr.write("CH2:PRObe %s" %(ch2probe))
            return

        def TVerticalScale(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Input a vertical scale for Channel 1 and 2 in a '1.0E0' format.")
            ych1=input("Channel 1 vertical scale?")
            self.instr.write("CH1:VOLts %s" %(ych1))
            ych2=input("Channel 2 vertical scale?")
            self.instr.write("CH2:VOLts %s" %(ych2))
            return

        def THorizontalScale(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Input the horizontal main axis scale in a '1.0E0' format.")
            xaxistime=input("Horizontal main axis scale?")
            self.instr.write("HORizontal:MAIn:SCAle %s" %(xaxistime))
            return

        def TTriggerType(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Select a trigger type like 'EDGE' (simple brackets included) and its settings.")
            trigg=input("Trigger type?")
            self.instr.write("TRIGger:MAIn:TYPe %s" %(trigg))
            triggch=input("Trigger channel is 'CH1' or 'CH2'?")
            self.instr.write("TRIGger:MAIn:EDGE:SOUrce %s" %(triggch))
            trigglev=input("Trigger level in a '1.0E0' format is?")
            self.instr.write("TRIGger:MAIn:LEVel %s" %(trigglev))
```

```
            triggslope=input("Trigger slope is 'RISE' or 'FALL'?")
            self.instr.write("TRIGger:MAIn:EDGE:SLOpe %s" %(triggslope))
            triggcoupl=input("Trigger coupling is 'DC' or 'AC'?")
            self.instr.write("TRIGger:MAIn:EDGE:COUPLing %s" %(triggcoupl))
            return

        def TTriggerState(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Event handler 'TTriggerState' not implemented!")
            triggstat=input("Osciloscope state is Run/Stop 'RUNSTOP' or Trigger
'SEQUENCE'?")
            self.instr.write("ACQuire:STOPAfter %s" %(triggstat))
            startstat=input("Trigger state is ready '1' or not ready '0'?")
            self.instr.write("ACQuire:STATE %s" %(startstat))
            return

        def HAbout(self, event):    # wxGlade: MyFrame.<event_handler>
            print("Program developed for a TFG. UPC ESEIAAT 10/6/19")


# end of class MyFrame

class ElementalApp(wx.App):
        def OnInit(self):
            self.frame = ElementalFrame(None, wx.ID_ANY, "")
            self.SetTopWindow(self.frame)
            self.frame.Show()
            return True

if __name__ == "__main__":
        app = ElementalApp(0)
        app.MainLoop()
```